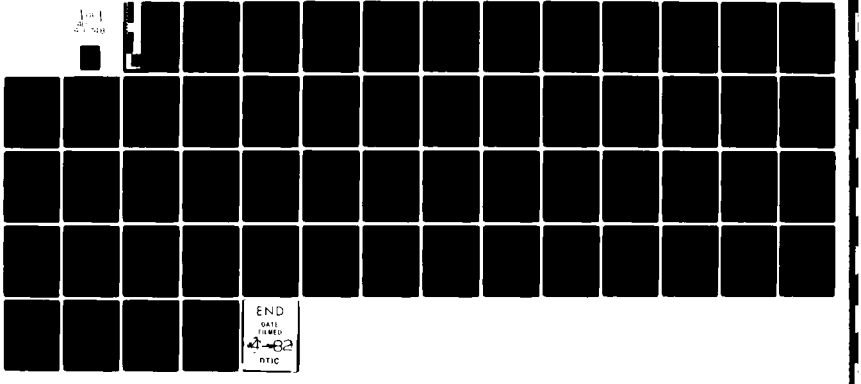
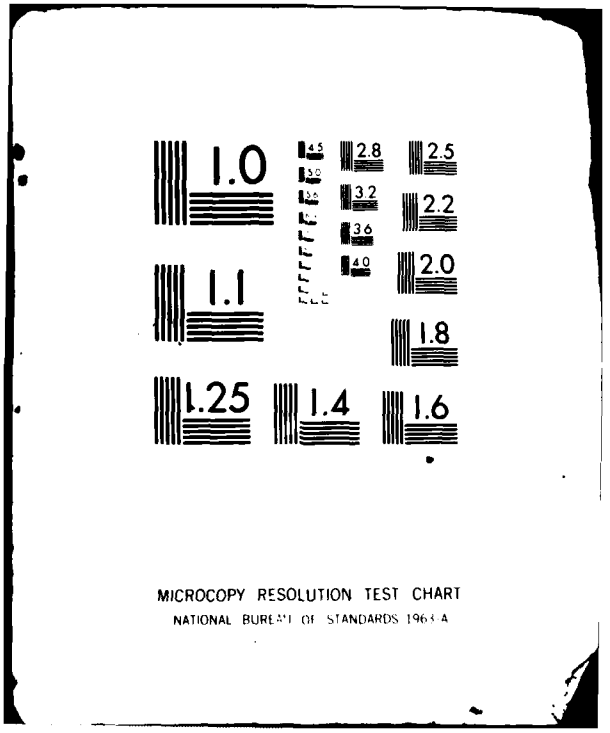
AD-A111 748 OHIO STATE UNIV COLUMBUS COMPUTER AND INFORMATION SC--ETC F/G 9/2
A SURVEY OF PARALLEL SORTING ALGORITHMS.(U)
DEC 81 D J DEWITT, D FRIEDLAND, D K HSIAO N00014-75-C-0573
UNCLASSIFIED OSU-CISRC-TR-81-11 NL



END
DATE FILMED
4-82
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

TECHNICAL REPORT SERIES

(12)

COMPUTER & INFORMATION SCIENCE RESEARCH CENTER

THE OHIO STATE UNIVERSITY COLUMBUS, OHIO

(OSU-CISRC-TR-81-11)

A SURVEY OF PARALLEL SORTING ALGORITHMS

by

David J. DeWitt and Dina Friedland[+]
Computer Sciences Department
University of Wisconsin
Madison, Wisconsin

David K. Hsiao and M. Jaishankar Menon*
Computer and Information Science Department
The Ohio State University
Columbus, Ohio

[+]This research was partially supported by the National Science Foundation under grant MCS78-01721 and the United States Army under contracts #DAAG29-79-C-0165 and #DAAG29-75-C-0024.

*This research was supported by the Office of Naval Research under contract N00014-75-C-0573.

Computer and Information Science Research Center

The Ohio State University

Columbus, OH 43210

December 1981

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER OSU-CISRC-TR-81-11	2. GOVT ACCESSION NO. AD-A111 748	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) "A Survey of Parallel Sorting Algorithms"		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) David J. DeWitt, David K. Hsiao Dina Friedland, M. Jaishankar Menon		8. CONTRACT OR GRANT NUMBER(s) N00014-75-C-0573
9. PERFORMING ORGANIZATION NAME AND ADDRESS Office of Naval Research Information Systems Program Arlington, Virginia 22217		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 4115-A1
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE December 1981
		13. NUMBER OF PAGES 52
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Scientific Officer	DDC New York Area
ONR BRO	ONR 437
ACO	ONR, Boston
NRL 2627	ONR, Chicago
ONR 1021P	ONR, Pasadena

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Internal parallel sorting algorithms, external parallel sorting algorithms, time complexity, memory requirements, I/O requirements, parallel algorithms vs. serial algorithms.

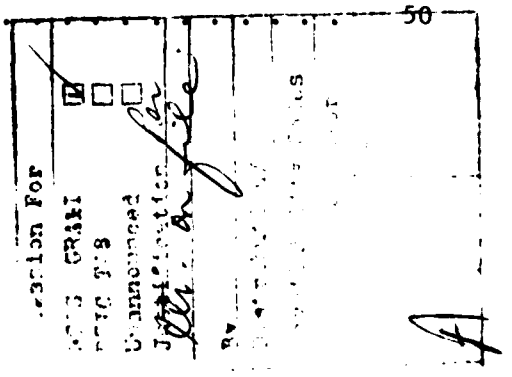
20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A rather comprehensive survey of parallel sorting algorithms is included herein. Parallel sorting algorithms are considered in two major categories - the internal parallel sorting algorithms and the external parallel sorting algorithms. Because external sorting algorithms are important to the database applications, considerable emphases are made in the motivation and analysis of the external parallel sorting algorithms surveyed in the report. In particular, the authors of this report have conducted research in external parallel sorting algorithms and made some important contributions. Their findings are also reported herein.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

TABLE OF CONTENTS



LIST OF FIGURES

PREFACE

This work was supported by a grant MCS78-01721 from National Science Foundation and by contracts DAAG29-79-C-0165 and DAAG29-75-C-0024 from the United States Army to Dr. David J. DeWitt (Associate Professor of Computer Science, University of Wisconsin) and by Contract N00014-75-C-0573 from the Office of Naval Research to Dr. David K. Hsiao (Professor of Computer and Information Science, Ohio State University). This report is prepared in the Laboratory for Database Systems Research of the Ohio State University which is funded by the Digital Equipment Corporation (DEC), Office of Naval Research (ONR) and the Ohio State University (OSU) and consists of the staff, graduate students, undergraduate students and faculty for conducting research in database systems. The ONR research contract was administered and monitored by the Ohio State University Research Foundation. This report is issued by the Computer and Information Science Research Center (CISRC) of the Ohio State University.

1. INTRODUCTION

Parallel algorithms have attracted considerable attention in recent years. They have been developed for various numerical problems known to consume large amounts of processing time on a serial computer. Because of their intrinsically parallel nature, certain classical numerical methods can be very efficiently adapted to a multiprocessor environment. In particular, various parallel algorithms have been developed for matrix multiplication ([Chan76]) and inversion ([Csan76], [Prep78b]), Fourier transforms ([Peas68], [Batc68]) and finite elements methods for partial differential equations ([Flan77], [Rose69]). The parallel computer architectures for which these methods are applicable include vector computers such as the Cray-1 ([Russ78]), as well as interconnection networks and mesh-connected processors such as the Illiac IV or the DAP ([Flan77]). In this paper, we present, however, a survey of parallel algorithms developed for sorting.

Sorting is defined as the process of rearranging a sequence of items into ascending or descending order. A basic sorting operation deals with items which are all key, that is the order is defined on the values of the items themselves. A more general sorting procedure deals with records where one field or the concatenation of several fields constitute the key with which the records are to be sorted. In a database environment, sorting refers to record sorting and this has significant implications in terms of storage and data movement, since typically a record will have several hundreds bytes, while the key or sort attribute may only be a few bytes long.

The purpose of sorting in a database environment is to facilitate access to a record set. It is more efficient to access a particular record in a sorted set than in an unsorted set and it is easier to identify duplicate records, since sorting brings them together. Sorting is also an efficient method for realizing the equijoin of two relations in a relational database. For example, if the tuples (i.e., records) of the relations (i.e., record sets) to be joined are presorted, the join can be performed in a single sequential pass over the relations.

Sorting appears to be a serial process by nature. However, the motivation for studying parallel sorting schemes is strong because of the high processing costs involved. Also, the complexity of parallel sorting algorithms is a real challenge to the theoretician and the introduction of parallel processing adds a new perspective to a classical area of research.

While fast algorithms have been developed for sorting on a uniprocessor, there is a fundamental difference in the approach taken for sorting an array and for sorting a file. In the case of array sorting, one assumes that the entire array is brought into a contiguous area of the processor's memory and that at the end of the sorting operation the sorted sequence also resides in the processor's memory. For this reason, array sorting is usually called internal sorting. The performance of an array sorting algorithm is measured by its execution time and its storage requirement. Depending on the size of the problem, a fast algorithm which requires a large amount of internal storage might be less desirable than a slower algorithm with smaller storage requirements. Execution time is determined by the number of comparisons and by the number of moves (or interchanges) required. Also, an analysis of a serial sorting method usually includes both the worst-case and the average-case execution time of the algorithm, since the number of comparisons and moves needed may depend on the initial distribution of the array elements. For example, it is known that the "Quicksort" algorithm has the best average behavior among commonly used algorithms while tree methods such as "Heapsort" perform better in the worst case. A "straight" sorting method would compare every key to every other key and therefore would require $O(n^2)$ comparisons. However the theoretical lower bound for the number of comparisons is nlogn ([Knu73]) and many sorting algorithms indeed achieve it.

File sorting algorithms are termed external sorting algorithms which require different evaluation criteria. Since a large file cannot fit in the main memory, an external sorting method must read some records of the file from the secondary storage (such as disks or tapes), process them, and write them back before some other records may be processed. Because every record must be compared to every other record, sorting cannot be performed in a single pass over the file and I/O costs become a significant factor in the design and evaluation of an external sorting scheme. Merging of sorted lists of records is the basic building block for external sorting. The simplest method is a 2-way merge and sort which consists of iteratively merging pairs of lists of k records into a sorted list of 2k records. Variations of this scheme include the n-way merge and sort (where n lists are merged together at each step) and the N-way Balanced Merge (where the output are written alternately on the different secondary devices). In practice, one does not use a pure merge-sort algorithm. Typically, the file is initially partitioned into equal sections small enough to fit in the main memory and these sections are sorted using a fast internal sorting

algorithm. When these sections have been sorted, the merge-sort procedure is initiated.

With one exception (Even's parallel tape sorting algorithm [Even74]) all previously published algorithms for parallel sorting are array sorting algorithms. They may also be classified as internal sorting algorithms, since they deal with the problem of sorting an array stored in the processors' memories (in most cases the array is distributed so that a single array element resides in each processor's local memory). Parallel processing makes it possible to perform more than a single comparison during each time unit. Some models (the sorting networks in particular) assume that a key is compared to only one other key during a time unit. This is restrictive but does not really limit the amount of parallelism because, in general, there are less processors available than pairs to compare. Another possibility is to compare a key to many other keys simultaneously. For example, in [Mull75], a key is compared to (n-1) other keys in a single time unit using n processors.

Parallelism may also be exploited to move many keys simultaneously. After a parallel comparison, the processors may exchange data. The concurrency which can be achieved in the data exchange is limited either by the interconnection scheme between the processors, if one exists, or by memory conflicts, if the shared memory is used for communication.

The analog to a comparison-and-move step in a uniprocessor memory is a parallel-comparison-and-concurrent-exchange step in a parallel organization. Therefore, it is natural to measure the performance of parallel sorting algorithms by the number of parallel comparison and concurrent exchanges required. Thus, the speedup of a sorting algorithm due to parallelization may be defined as the ratio between the number of comparison-moves required by an optimal serial sorting algorithm and the number of comparison-exchanges required by the parallel algorithm. Since an optimal serial algorithm sorts n keys in O(nlogn) time, an optimal speedup via parallelization would occur when n keys are sorted with n processors in time O(logn). However, it does not seem possible to achieve this bound by simply parallelizing one of the well-known optimal sorting algorithms, since it appears that the best serial sorting algorithms have severe serial constraints which cannot be removed. On the other hand, parallelization of straight sorting methods (i.e., brute-force methods requiring $O(n^2)$ comparisons) seems easier but it cannot lead to very fast parallel algorithms. By performing n comparisons instead of one in a single time unit, the execution time can be reduced from $O(n^2)$ to O(n). Partial parallelization of a fast serial

algorithm can also lead to a parallel algorithm of order O(n). For example, if we assign a processor to each of the (2n-1) nodes of a selection tree, we can transfer the minimum key to the root processor in logn steps and in (n-1) additional sequential steps we can also transfer, in order, all the remaining keys to the root.

The speedup achieved with these simple parallelization schemes (logn for n processors) was not satisfactory and many efforts have been made to improve it. The first major improvement was reached by the sorting networks with a speedup of n/logn. Recently, Preparata has shown [Prep78a] that the optimal bound (time: O(logn), speedup: n) can be achieved by using a theoretical model of n processors accessing a large shared memory. Furthermore, Baudet has shown [Baud78] how the optimal speedup equal to the number of processors may be asymptotically achieved by using a single-instruction-stream-and-multiple-data-stream (SIMD) computer in which the number of elements to be sorted far exceeds the number of processors used to do the sorting. These and other internal sorting algorithms are described in Section 2.

A characteristic of many of the internal sorting algorithms [Nass79, Thom77, Ston71] is that they use p processors to sort p (or 2p) elements. Therefore, even though these methods are fast, they suffer from the fact that groups of elements larger than p (or 2p) in number will have to be sorted in separate batches of p (or 2p) elements and then be merged. Thus, these methods are processor-limited, i.e., the number of elements that can be sorted is limited by the number of processors. Similarly, we conclude that the methods of [Prep78a, Hirs78] are also processor-limited. The methods presented in [Baud78], however, are not processor-limited. Thus, in these methods, p processors are used to sort Mp elements, where each processor has enough primary memory to store M elements.

Internal sorting algorithms, however, do not address the problems of sorting a large data file in a database environment in which the number of records to be sorted is significantly larger than the available memory of the multiprocessor. In Section 3, we present a description of several external sorting algorithms which have been developed to overcome the problems. Included are Even's parallel tape sorting algorithms along with several external algorithms which have been developed by the authors. Finally, a summary and suggestions for future research are presented in Section 4.

2. INTERNAL SORTING ALGORITHMS

Research on parallel algorithms for internal sorting can be divided into three broad categories. The earliest algorithms use sorting networks. A set of n numbers entering a sorting network on n separate input lines is sorted by performing a sequence of parallel comparisons and concurrent exchanges on pairs of numbers. The best sorting networks achieve a speed of $O(\log^2 n)$. That is, they can sort n numbers in approximately $\log^2 n$ comparisons and exchanges. A second category of algorithms assumes a model where the processors have more complexity than comparators modules and they share a large common memory. The type of sorting algorithm used for this model is very different from the sorting networks. Most of them are "enumeration" algorithms which associate with each element in the array being sorted a count specifying the element position in the sorted array. Optimal algorithms in this category are very fast and achieve the optimal lower bound of O(logn). A third category of algorithms assumes that each processor has enough local memory to store M keys. Thus, a total of Mp keys may be sorted by p processors. These algorithms are not processor-limited and are optimal when M>>p.

In the following sections, we describe these three models and also discuss parallel sorting on a mesh-connected multiprocessor [Thom74, Nass79].

2.1 Sorting Networks

One of the earliest results in parallel sorting is due to Batcher, who developed two methods to sort n keys with $O(n\log^2 n)$ comparators in time $O(\log^2 n)$. A comparator is a module which receives two keys on its two input lines A, B and outputs the lower key on its higher output line L and the higher key on its lower output line H (see Figure 1). A serial comparator receives A and B with their most significant bits first and can be realized with a small number (say, 13) of NOR gates put on an integrated circuit chip. Parallel comparators where several bits are compared in parallel at each step are faster but obviously more complex. Batcher's algorithms, odd-even merges and bitonic sorts, are all based on the principle of iterated merging. Starting with an initial sequence of 2^k keys, a specific iterative rule is used to create sorted lists of 2, 4, 8, ... ,2^k keys during successive stages of the algorithm.

2.1.1 Odd-Even Merges

The iteractive rule for an odd-even merge is illustrated in Figure 2. Given two sorted sequences of keys $(a_1, a_2, \ldots)$ and $(b_1, b_2, \ldots)$, two new se-

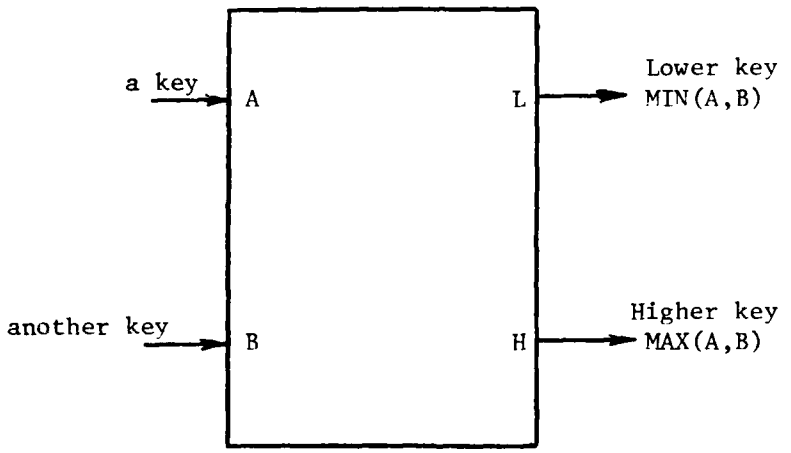


Figure 1. A Comparison-Exchange Module

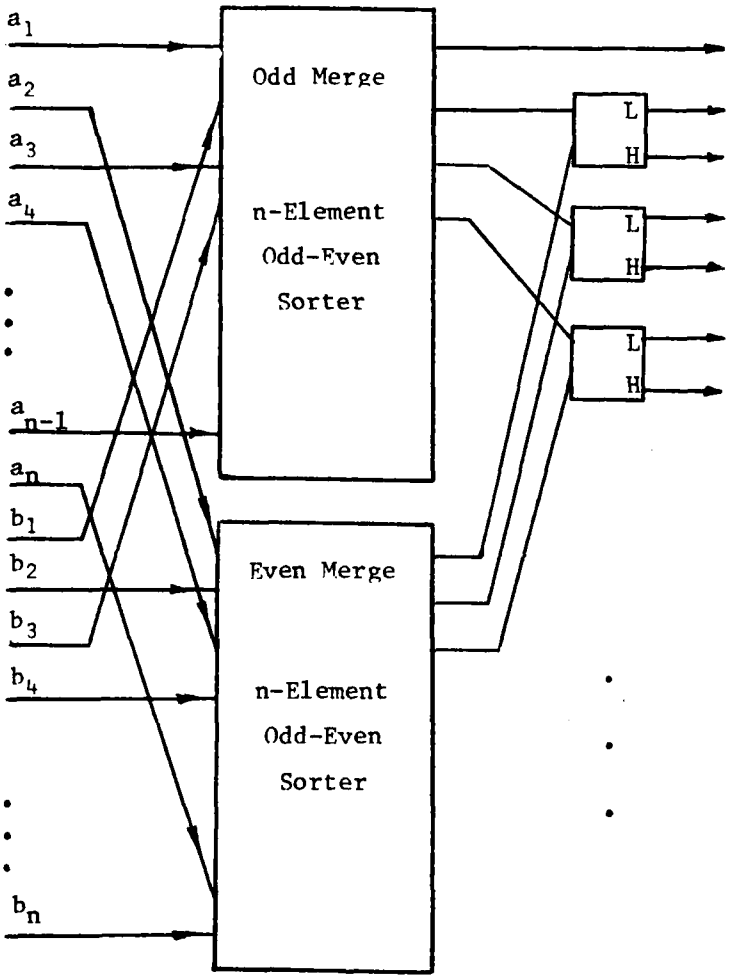


Figure 2. The Iterative Rule for the Odd-Even Sort

quences ("odd" and "even" sequences) are created: one consists of the odd-numbered keys and the other of the even-numbered keys from both sequences. The odd sequence $(c_1, c_2,...)$ is obtained by merging the odd terms $(a_1, a_3,...)$ with the odd terms $(b_1, b_3,...)$. Similarly, the even sequence $(d_1, d_2,...)$ is obtained by merging the even terms $(a_2, a_4,...)$ with the even terms $(b_2, b_4,...)$. Finally the sequences $(c_1, c_2,...)$ and $(d_1, d_2,...)$ are merged into $(e_1, e_2,...)$ by applying the following comparison-exchanges:

$$e_1 = c_1$$

$$e_{2i} = \max(c_{i+1}, d_i)$$

$$e_{2i+1} = \min(c_{i+1}, d_i)$$

The resulting sequence will be sorted. (For a proof the reader is referred to [Knut73], pg. 224,225.)

To sort 2^k keys using the odd-even iterative merge requires 2^{k-1} 2-by-2 merging networks, followed by 2^{k-3} 4-by-4 merging networks, and so on. Since 2^{i+1}-by-2^{i+1} merging network requires one more step of comparison-exchanges than a 2^i-by-2^i merging network, it follows that an input key goes through at most $k(k+1)/2$ comparators (since $1+2+...+k = k(k+1)/2$). This means that 2 keys are sorted by performing $k(k+1)/2$ parallel comparisons and exchanges. However, the number of comparators required by this type of sorting network is $((k^2-k+4)2^{k-2}-1)$ [Batc68]. Several subsequent works ([Knut73]) have been able to reduce this number of comparators, but only in some particular cases (e.g., $k \leq 4$).

2.1.2 Bitonic Sorts

A bitonic sequence is obtained by juxtaposing two monotonic sequences into one when one montonic sequence is ascending and the other descending[1]. For example, the following bitonic sequence consists of an ascending monotonic sequence of (3589) and a decending monotonic sequence of (6421).

3 5 8 9 6 4 2 1

For a bitonic sort, a second iterative rule is used (see Figure 3). The iterative rule is based on the observation that a bitonic sequence may be split into two bitonic subsequences by performing a single step of comparison-exchanges. Let $(a_1, a_2,...,a_{2n})$ be a bitonic sequence such that $a_1 \leq a_2,..., \leq a_n$ and

[1] A more general definition of a bitonic sequence allows a cyclic shift of such a sequence, e.g., 89642135 would also be a bitonic sequence.

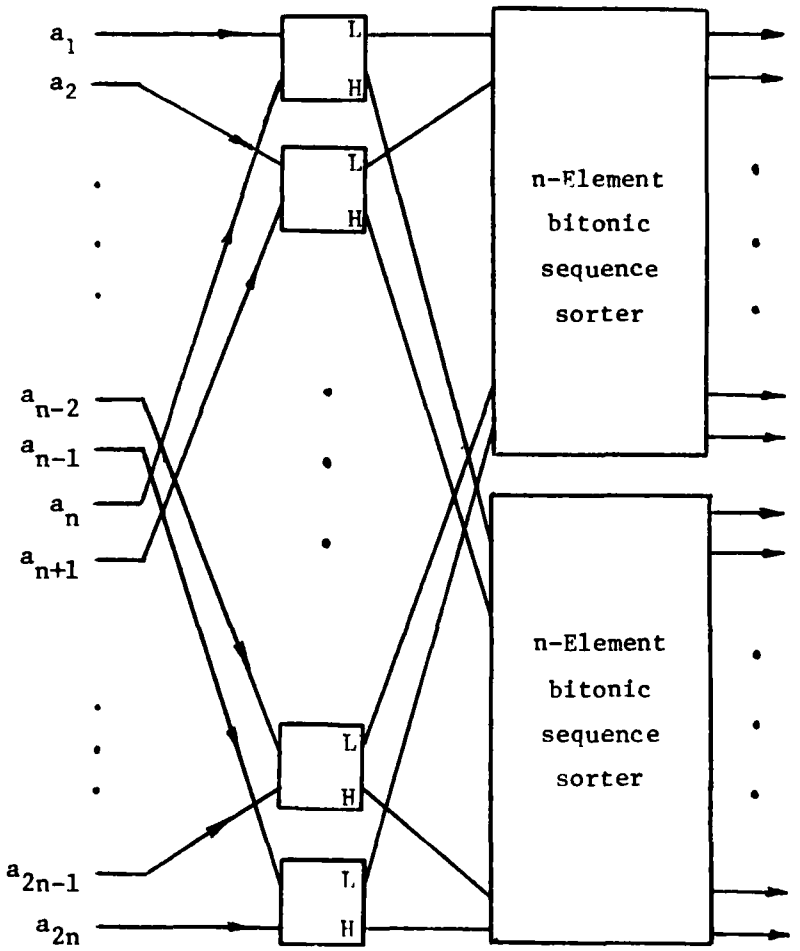


Figure 3. The Iterative Rule for the Bitonic Sort

$a_{n+1} \geq a_{n+2} \geq \ldots \geq a_{2n}$. Then the subsequences

$$\min(a_1,a_{n+1}),\ \min(a_2,a_{n+2}),\ldots,\ \min(a_n,a_{2n})$$

and

$$\max(a_1,a_{n+1}),\ \max(a_2,a_{n+2}),\ldots,\ \max(a_n,a_{2n})$$

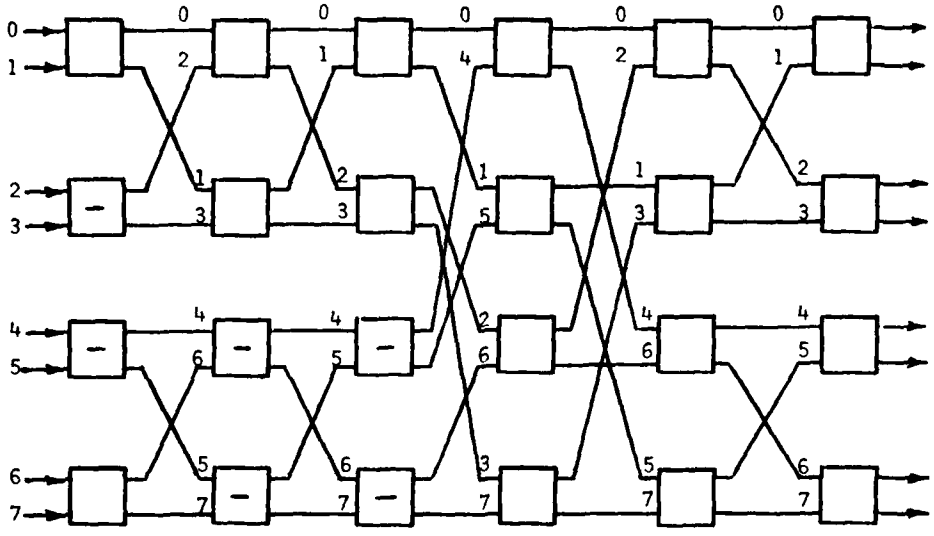
are both bitonic. Furthermore, the first subsequence contains the n lowest keys of the original sequence while the second contains the n highest. It follows that a bitonic sequence can be sorted by sorting separately two bitonic subsequences which are one half as long.

To sort 2^k keys using the bitonic iterative rule, we can successively sort and merge smaller sequences into larger sequences until a bitonic sequence of size 2^k is obtained. This sequence can be split into "lower" and "higher" bitonic subsequences. Note that the recursive-building procedure of a bitonic sequence must use some comparators which invert their output lines in order to output a pair of keys in decreasing order. This is necessary in order to build the decreasing subsequence of a bitonic sequence (see Figure 4). A bitonic sort of 2^k keys requires $k(k+1)/2$ steps, each using 2^{k-1} comparators.

Since the first version of the bitonic sort was presented, the algorithm has been considerably improved by the introduction of the "perfect shuffle" interconnection [Ston71]. Stone noticed that if the inputs were labeled by a binary index, then the indices of every pair of keys that enter a comparator would differ by a single bit in their binary representations, and this bit would be the j-th bit on step i, where $j = [(i-1) \bmod k + 1]$, of the bitonic sort. On the other hand, "shuffling" the indices (in a manner similar to shuffling a deck of cards) is equivalent to circularly shifting their binary representation to the left. Shuffling twice would shift the binary representation of each index twice. Thus the bitonic sort can be executed using a single rank of 2^{k-1} processors which are connected with a set of shift registers and shuffle links as shown in Figure 5.

Stone's modified version of the bitonic sort can sort n keys with n/2 processors in $\log^2 n$ shuffle steps and $1/2(\log n)(\log n + 1)$ comparisons-exchanges. This provides a speedup of $O(n/\log n)$ over the $O(n \log n)$ complexity of serial sorting. Therefore, it improves significantly the previous known bound of $O(n)$ for parallel speedup with n processors. The algorithm is illustrated in Figure 6.

Sorting networks are characterized by their "non-adaptivity" property. They perform the same sequence of comparisons regardless of the result of



: Indicates a comparator which inverts its output lines

Figure 4. A Bitonic Sort for 8 Keys

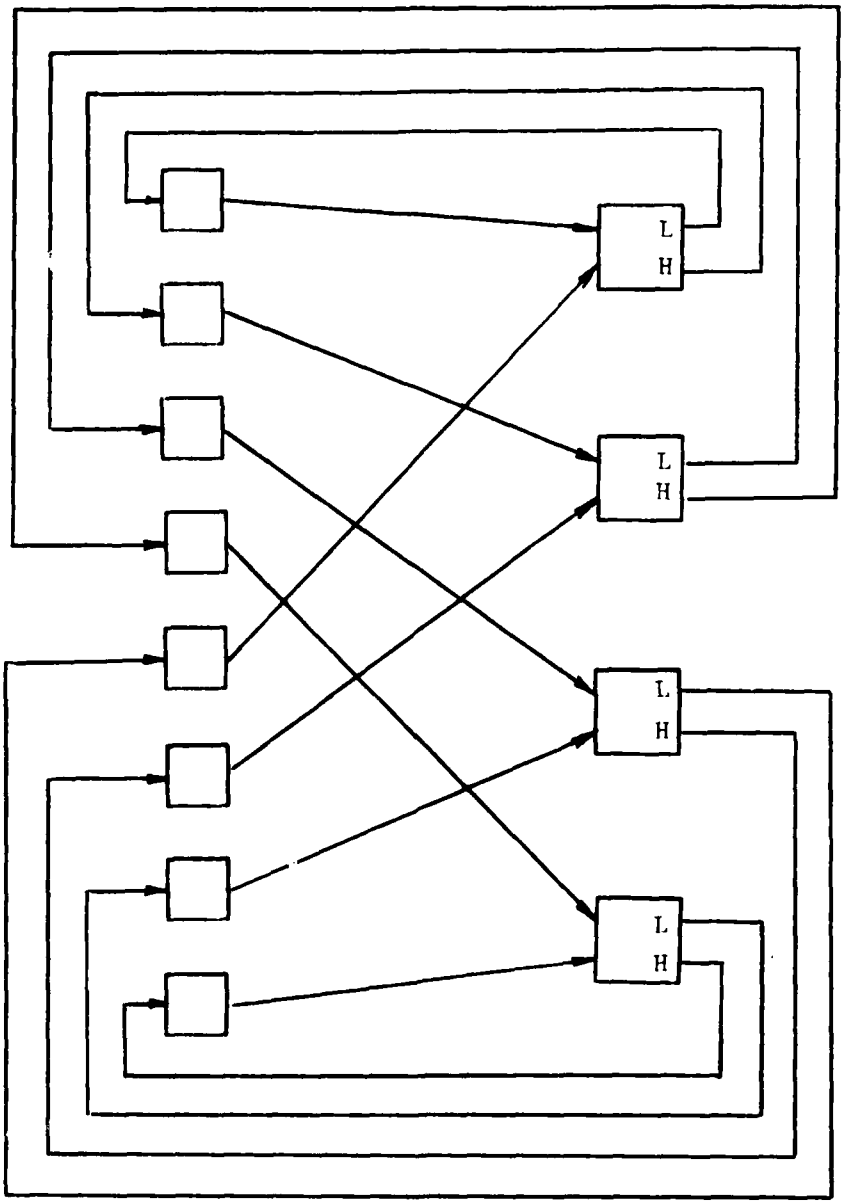


Figure 5. Stone's Architecture for a Bitonic Sorter

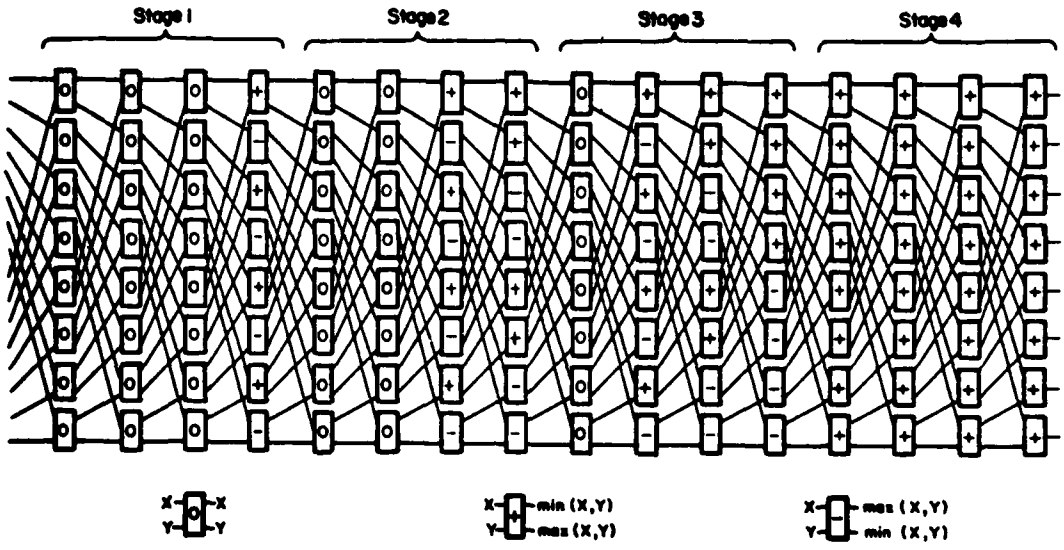


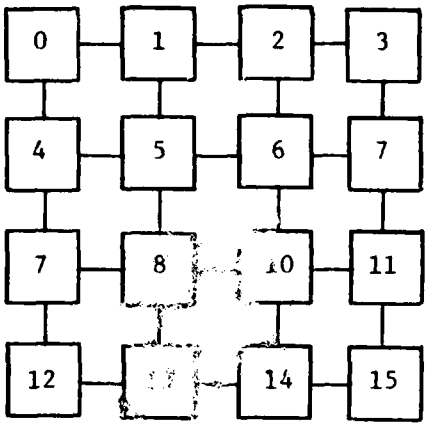
Figure 6. Bitonic Sort of 16 Keys With a Perfect Shuffle Connected Multiprocessor

intermediate comparisons. In other words, whenever two keys R_i and R_j are compared, the subsequent comparisons of R_i in the case $R_i < R_j$ are the same as the comparisons that R_j would have entered in the case $R_j \leq R_i$. The non-adaptivity property makes the implementation of an algorithm very convenient for an SIMD machine: the sequence of comparisons and transfers to be executed by all the processors is determined when the sorting operation is initialized. For example, it is shown [Sieg77] that the bitonic sort can be implemented on SIMD machines that have interconnection schemes different from the perfect shuffle.

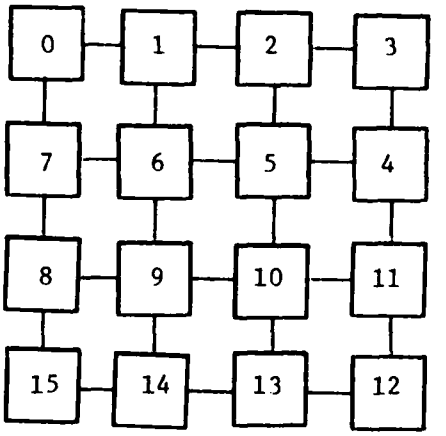
2.2 Bitonic Sorts on Mesh-Connected Processors

A different sorting problem is considered in [Thom75], in which the processors of an n-by-n mesh-connected multiprocessor are indexed according to a prespecified rule. The indexing rules considered are the row-major, the snake-like row-major and the shuffled row-major rules (shown in Figure 7). Assuming that n^2 keys with arbitrary values are initially distributed so that exactly one key resides in each processor, the sorting problem consists of moving the i-th smallest key to the processor indexed by i, for $i=1,\ldots,n^2$. As with the sorting networks, parallelism is used to simultaneously compare pairs of keys, and a key is compared to only one other key at any given unit of time. Concurrent data movement is allowed but only in the same direction, that is all processors can simultaneously transfer the contents of their transfer registers to their neighbors to the right, left, above or below. This computation model is SIMD since at each time unit a single instruction (compare or move) can be broadcast for concurrent execution by the set of processors specified in the instruction. The complexity of a method which solves the sorting problem using this model can be measured in terms of the number of comparisons and unit-distance routing steps. For the rest of this section we refer to the unit-distance routing step as a move. Any algorithm which is able to perform such a permutation will require at least 4(n-1) moves, since it may have to interchange the elements from two opposite corners of the array processor. This is true for any indexing scheme. In this sense a sorting algorithm which requires O(n) moves is optimal.

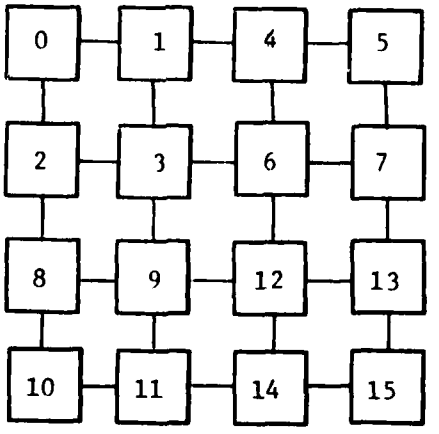
In [Thom75] two algorithms were presented which sort in O(n) comparisons and moves. The first algorithm uses an odd-even merge of two dimensional arrays and orders the keys with snake-like row-major indexing. The second uses a bitonic sort and orders the keys with shuffled row-major indexing. Recently, a third algorithm that sorts with row-major indexing with similar



Row-major indexing



Snake-like row-major indexing



Shuffled row-major indexing

Figure 7. Indexing Schemes

performance has been published [Nass79]. This algorithm is also an adaptation of the bitonic sort where the iterative rule is a merge of two dimensional arrays.

2.3 The Odd-Even Transposition Sort

The serial odd-even transposition sort [Knut73] works as follows. Let $(x_1, x_2, \ldots, x_n)$ be a set of keys to be sorted. There are n steps of comparison-exchanges: during the odd steps, $x_1, x_3, \ldots$ are respectively compared to $x_2, x_4, \ldots$ and exchanged, if $x_1 > x_2$, $x_3 > x_4, \ldots$. At the even steps, x_i and x_{i+1} (i=2, 4, 6,...) are compared and exchanged, if $x_i > x_{i+1}$.

This algorithm calls for a straightforward parallelization [Baud78]. Consider n linearly connected processors and label them $P_1, P_2, \ldots, P_n$. We assume the links are bidirectional so that P_i can communicate with P_{i-1} and with P_{i+1}. Initially x_i resides in P_i for i=1, 2,..., n. For a parallel sort, let odd numbered processors such as P_1 and P_3 be active during the odd steps and perform in parallel the odd steps of the serial odd-even transposition sort. Similarly let even numbered processors such as P_2 and P_4 be active during the even steps and perform in parallel the comparison-exchanges (x_1, x_{i+1}) for even i's. Note that a single comparison-exchange requires two transfers. For example, during the first step, x_2 is transferred to P_1 and compared to x_1 by P_1. Then, if $x_2 < x_1$, x_1 is transferred to P_2: otherwise, x_2 is transferred back to P_2. Therefore, the algorithm sorts n keys with n processors in n comparisons and 2n moves. This is not as fast as other algorithms we have previously described, but [Baud78] describes a generalization for the case where there are p processors and n=Mp keys to sort and he shows that the modified algorithm is optimal if $n \gg p$.

The idea is to distribute the sequence to be sorted equally among the p processors so that M keys are stored in each processor's local memory. At the end of the sorting procedure, processor P_i should have in its local memory a sorted sequence S_i of M keys. Moreover, the concatenation of these partial sequences $S_1 S_2 \ldots S_p$ should be a sorted sequence of length n. This is a natural extension of the definition of a sorted array stored in a single processor memory in the sense that one could consider the total address space of the linearly connected multiprocessor as the ordered concatenation of the processors local memories so that the first processor contains the first M addresses and the p-th processor contains the last M addresses. With M keys stored in each processor, the odd-even transposition sort may be generalized as follows. Initially each processor sorts internally its M keys, using a fast

serial algorithm. Then the algorithm proceeds as before provided that a comparison exchange step is replaced by a "merge-split" step, which consists of merging 2 sorted sequences each of which has M keys splitting the resulting sequence of 2M keys into two and transferring the higher half to the right neighbor processor. Merging two sequences requires 2M comparisons and 2M transfers are required for each merge-split step. Therefore, after the initial internal sort the algorithm requires 2Mp (i.e., 2n) comparisons and moves. The initial phase requires Mlog (i.e., (n/p) log (n/p)) comparisons and this term will determine the O(nlogn/p) complexity of the algorithm in the case that $p << \log n$.

2.4 Sorting on a Shared-Memory Multiprocessor

After the bound of $O(\log^2 n)$ was achieved through the use of sorting networks, considerable effort was devoted to improve this result and to achieve the theoretical bound of O(logn).

2.4.1 Sorting Network Methods

The first model that was able to reach this bound may be designated as a "modified" sorting network [Mull75]. Instead of comparison-exchange modules, this model uses comparators which input two keys A, B and output a single bit x(x = 0 if A < B or x = 1 if A > B). To sort a sequence of n keys, each key is simultaneously compared to all the others by using a total of n(n-1) comparators. The output bits from the comparators are then fed into a parallel counter which computes in logn steps the rank of a key by counting the number of bits set to 1 in the comparison of this key with all the other (n-1) keys. Finally, a switching network consisting of a binary tree of (logn+1) levels of single-pole, double-throw switches routes a key of rank equal to i to the i-th terminal of the tree. There is one such tree for each key, and each tree uses (2n-1) switches. Routing a key through this tree requires logn time units, and this step determines the algorithm complexity. A diagram for this type of sorting network is presented in Figure 8.

At the cost of additional hardware complexity (the basic modules are more complex than comparison-exchange modules and the network uses more of them), the above algorithm sorts n keys in O(long) time with $O(n^2)$ processing elements. This result was the first to use an enumeration scheme for parallel sorting. Later algorithms which we refer to as "enumeration type" sorting algorithms exploit the same idea.

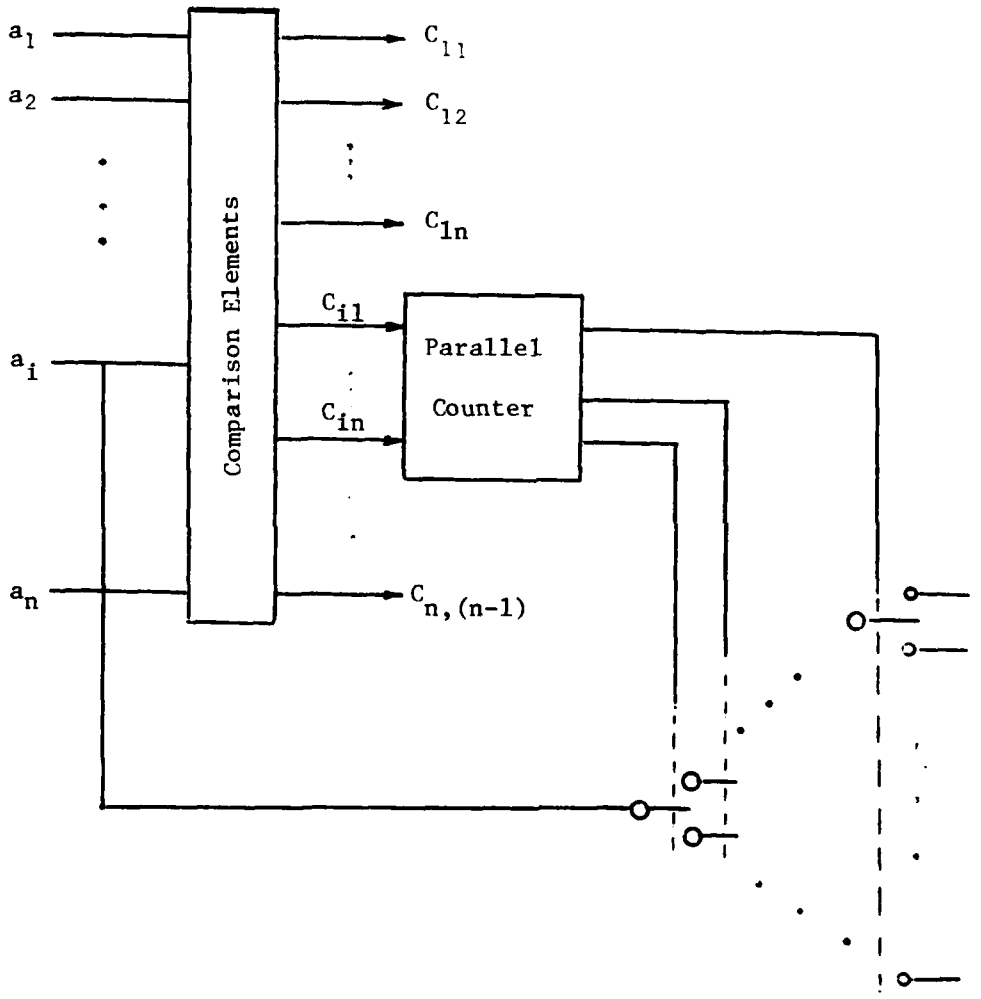


Figure 8. Sorting Network not Constructed of Comparison-Exchange Modules

The sorting network model and its modified version were embedded in a more general multiprocessor model where n processors have access to a large shared memory. With this scheme, sorting is performed by computing in parallel the rank of each key (the "enumeration" phase) and routing the keys to the location specified by their rank. The algorithm we have just described performs the enumeration with n(n-1) comparators and the routing with n trees of (2n-1) switches each. Therefore, it can also be described as an enumeration-type algorithm which sorts n keys in time O(logn) on a multiprocessor which consists of $O(n^2)$ processors sharing a common memory of $O(n^2)$ cells.

A major improvement of this algorithm calls for a reduction in the number of processors. Even from a pure theoretical point of view, the requirement of n^2 processors for achieving a speed of O(logn) is discouraging. An optimal parallel sorting algorithm should achieve the same speed with only O(n) processors in order to show a speedup of order n.

2.4.2 Faster Parallel Merge Methods

In a study [Vali75] of parallelism in comparison problems, an algorithm with a shared memory model that merges two sorted sequences of n keys and m keys (where $n \leq m$) with $\sqrt{nm}$ processors in 2loglogn+O(1) comparison steps (compared to logn for the bitonic merge) has been proposed. On the other hand, the problem of merging two sorted sequences of n keys and m keys with a smaller number of processors p ($p \leq n \leq m$) has been studied. By employing a simple parallel binary insertion, an algorithm [Gavr75] solves this problem with (2log(n+1)+4n/p) comparisons when two sorted sequences have the same number of keys. These two fast merge procedures were the basis for subsequent parallel sorting slgorithms ([Hirs78], [Prep78a]) of optimal complexity O(logn).

2.5 Bucket Sorts

A bucket sort algorithm [Hirs78] which sorts n keys with n processors in time O(logn), provided that the key values are in the range {0, 1,.. ,m-1} where m is termed the bucket number. A side effect of this algorithm is that duplicate keys are eliminated. It would be sufficient to have m buckets and to assign one key to each processor (the processor that gets the i-th key would be labeled P_i); P_i would then place the value i in the appropriate bucket. For example, if P_3 had the 3rd key which is 5, it would place the value 3 in bucket 5. In other words, key values become bucket addresses. However, with this simplistic solution a bucket memory conflict may result

when several processors attempt simultaneously to store different values of i in the same bucket. The memory contention problem may be solved by increasing substantially the memory requirements. Suppose there is enough memory available to create n arrays for each bucket. Each processor (there are n of them) can then mark a bucket without any fear of memory conflict. To complete the enumeration sort the n arrays of a bucket must be merged. This is done by using a sophisticated parallel merge procedure, where processors are granted simultaneous read access to a memory location but no write conflict can occur. Hierschberg generalizes the above method so that duplicate keys remain in the sorted array. But this degrades the performance of the sorting algorithm. The result is a method which sorts n keys with $n^{1+k(1/k)}$ processors in time O(klogn), where k is an arbitrary integer [Hirs78].

A major drawback of this algorithm (aside from the lack of realism of the shared memory model which will be discussed later) is its m*n space requirement. Even when the range of possible key values is not very large, one would like to reduce this requirement. In the case of a wide range of key values (for example if the keys are character strings rather than integer numbers), the algorithm would not be applicable.

2.6 Sorting By Enumeration

In [Prep78a] two new fast sorting algorithms are presented, which are both enumeration-type algorithms. However, rather than computing separately the rank of every single key, they first partition the keys into a number of key lists, sort the key lists and compute their ranks by merging pairs of key lists. Finally, for each key the sum of its key list ranks is also computed in parallel.

The first algorithm uses Valiant's merging procedure [Vali75] and sorts n keys with nlogn processors in time O(logn). The second algorithm uses Batcher's odd-even merge and sorts n numbers with $n^{1+(1/k)}$ processors in time O(klogn). The latter algorithm performs like Hierschberg's but it has the additional advantage of being free from memory contention. Recall that Hirschberg's model required simultaneous fetches from the shared memory, while Preparata's method does not, since each key participates in only one comparison at any given unit of time.

Despite efforts made by these authors to eliminate memory conflicts in both bucket sorts and enumeration sorts, they are still not very realistic. Any model requiring at least as many processors as the number of keys to be sorted, all sharing a very large common memory, is not feasible with present or near-

term technology. However, the results achieved are of major theoretical importance and the methods used demonstrate the intrinsic parallel nature of certain sorting procedures. Furthermore, it seems that many of the basic ideas in these algorithms can inspire the design and the implementation of realistic parallel sorting methods for multiprocessors. For example, in Section 3.2.3 we present a simple method for parallel sorting by enumeration, which we plan to implement on a backend multiprocessor. While the efficiency of this method relies on the assumption that a fast broadcast facility is available, no shared memory is required.

3. EXTERNAL SORTING ALGORITHMS

In this section we present several parallel algorithms for sorting files. These algorithms differ from key sorting algorithms presented in Section 2 in several respects. First, they are designed to sort files for which the entire records of the files must reside in the prime memory. An internal sort algorithm only requires the keys of the records to be present in the prime memory. The memory and input line requirements of external sorting algorithms far exceed the memory and input line requirement of the internal sorting algorithms. A second difference between these two classes of parallel algorithms is the criteria by which different algorithms are evaluated. For internal sorting algorithms, the primary evaluation criteria are the number of comparisons and moves required to put the key in the sorted order. For external sorting algorithms, while the number of comparisons and moves required are important, the principal criterion is the number of input/output operations required. Thus, none of the algorithms presented below can be considered as optimal if our principal criterion is in terms of the number of key comparisons required.

We begin with three external merge-sorting algorithms including Even's parallel tape sorting algorithm [Even76]. Then, in the following section, we describe two different classes of block sorting algorithms. These algorithms are generalizations of non-adaptive sorting algorithms which use only comparison-exchanges and are obtained by replacing all comparison-exchange steps by merge-split steps. Two different techniques for merge-splitting are available, and these lead to the two different classes of block sorting algorithms.

3.1 Terminology and Notation

To facilitate our discussion, we introduce some terminology. We make the simplifying assumption that records are of fixed-length. Furthermore, let us

suppose that one page of memory can hold exactly one record. These two assumptions simplify the graphical illustration of the algorithm but are not essential to their understanding or operation. Each record is composed of a number of attribute-value paris. Whenever a record set is to be sorted, it is meant that the records of the set are ordered on the basis of increasing (or decreasing) values of a certain attribute (i.e., key) of the set. Thus, to sort a record set, an attribute (key) of the set must be designated first as the sort attribute. The sort values are those attribute values of the sort attribute. The number of sort values (not all of them being distinct) is equal to the cardinality of the record set. For example, in Figure 9, we illustrate a set of three records each of which consists of attribute-values on rank, age, salary, etc. We may sort these records on a number of attributes. If the chosen sort attribute is Rank, then we may represent the records with their rank values, namely, 5 for Record 1, 3 for Record 2 and 9 for Record 3. We do not have to show other attribute values of the records in representation, because these other attribute values are not in consideration. (See Figure 9 again). Upon sorting, Record 1 should precede Record 3 and follow Record 2, since this is the sequence of the sort values. On the other hand, if we choose either Age or Salary as the sort attribute, the sequence of sorted records will be different. More specifically, Record 1 will be last and Record 3 will be first in the sequence. This sequence of records is dictated by either ages or salaries. (see Figure 9 once more). This example shows that we represent records with their sort values. For simplicity, we eliminate the other attribute values. In the following sections, we shall use positive integers to represent the sort values of the records. A large rectangle represents a sorted run of records in one or more pages. Each page is represented by a small rectangle. The integer within the page denotes a record. Furthermore, we shall use circles to denote the processors. In Figure 10 for example, there are four processors and 16 pages.

To aid in the analyses of these algorithms, we let

- n: the number of records to be sorted
- p: the number of processors
- Cr: the time to read a page (from the secondary memory or cache)
- Cw: the time to write a page (into the secondary memory or cache)
- Cm: the time to merge two pages into a sorted run of two pages
- C_p^1: $Cr + Cm + Cw$ C_p^2: $2Cr + Cm + 2Cw$ C_p^3: $2Cr + Cm + Cw$
- M: n/p
- Cs: time to send a page from one processor to another

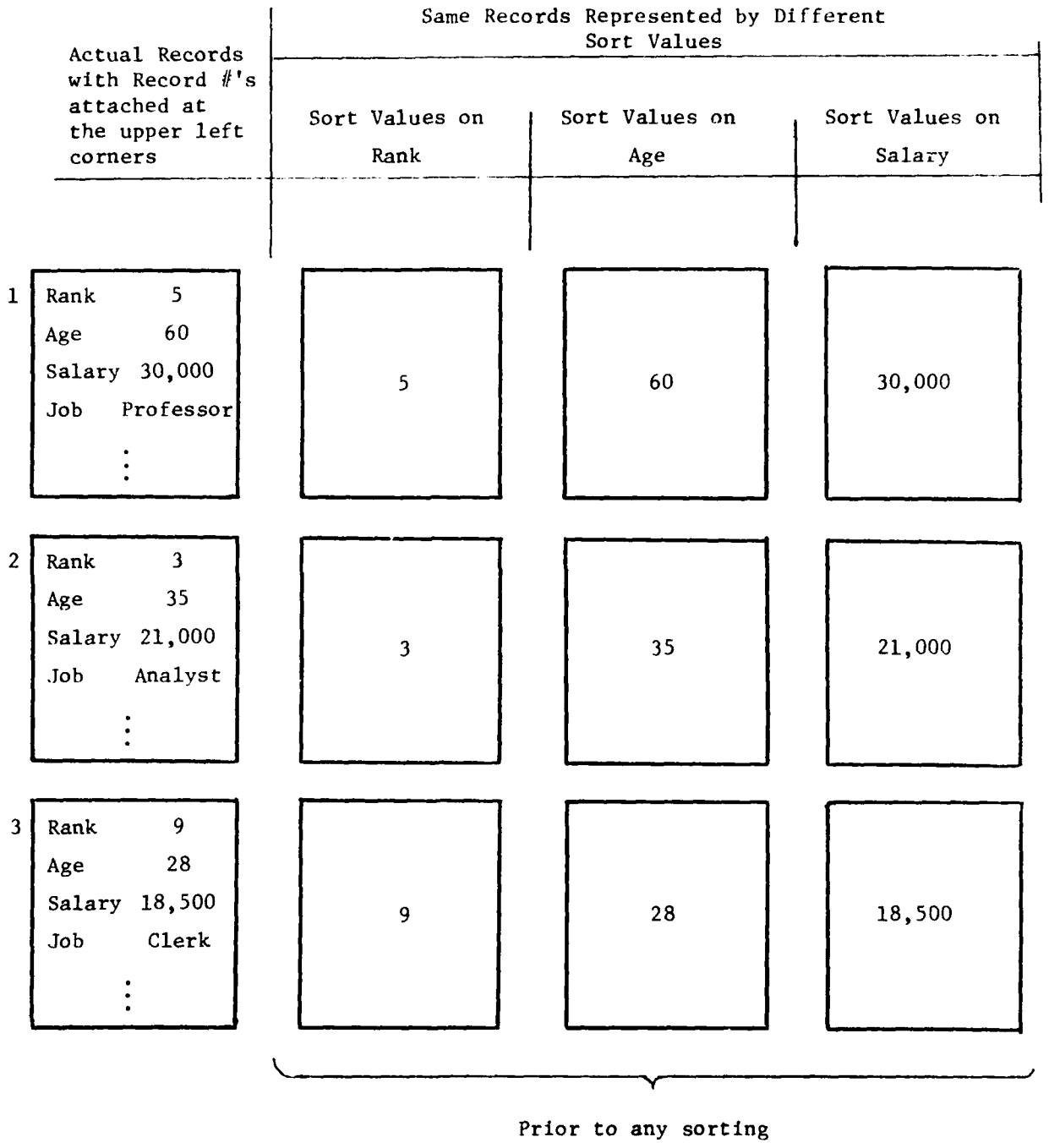


Figure 9. Representing Records by Sort Values

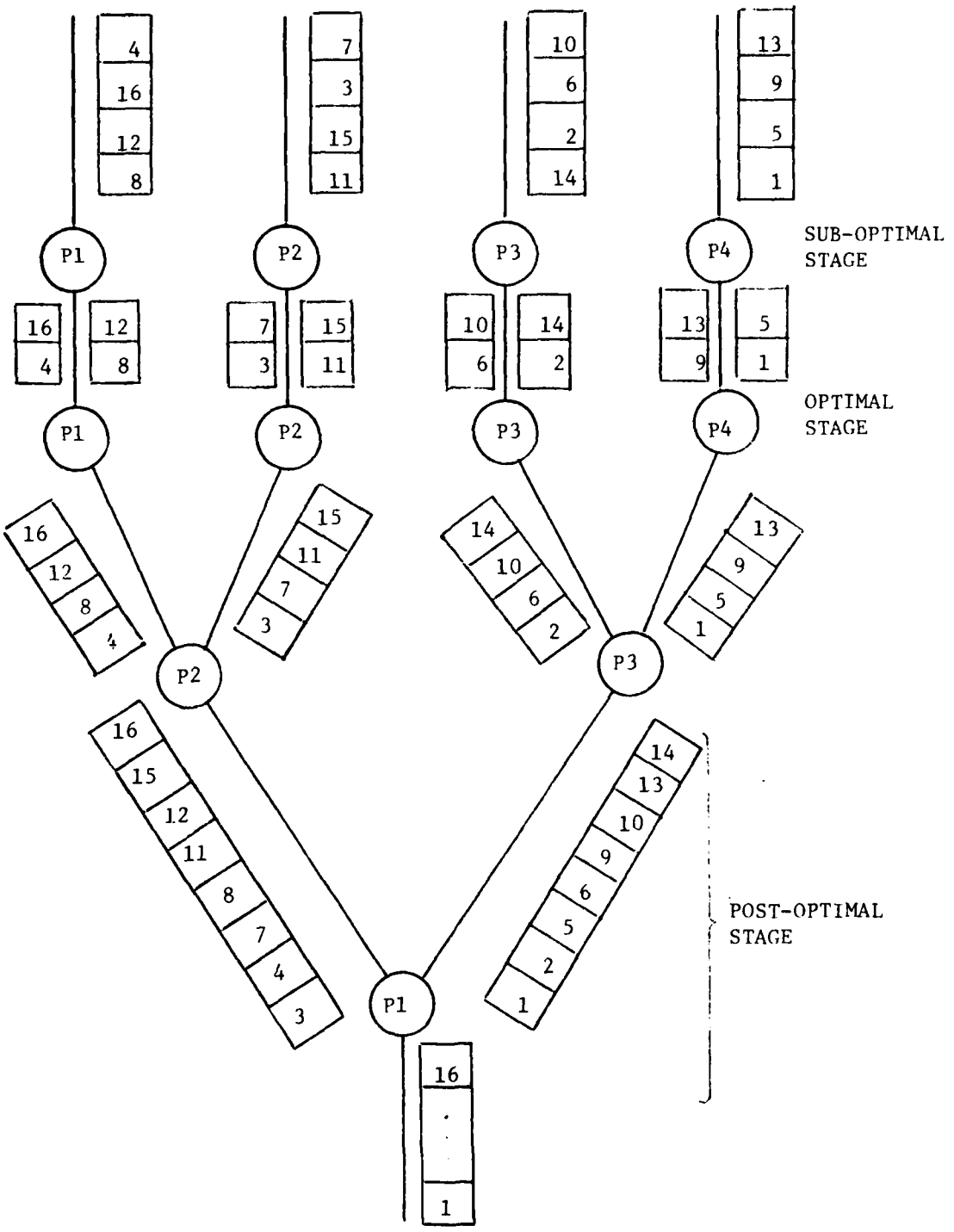


Figure 10. Parallel Binary Merge with 4 Processors and 16 Pages

3.2 External Merge-Sorting Algorithms

In this section we describe three parallel algorithms which are based on the traditional serial approach for externally sorting a file, i.e., the external merge-sort.

3.2.1 Even's Tape Sorting Algorithm

In [Even74], two parallel tape sorting algorithms are presented which are considered to be the first contribution to parallel external sorting algorithms. The sorting problem addressed in [Even74] is to sort a file of n records with p processors, where $p << n$. The only internal memory requirement is that three records could fit simultaneously in one processor's local memory. Both algorithms are parallel versions of an external 2-way merge-sort. They sort a file of n records by merging iteratively pairs of sorted runs of size 2, 2^2,..., $2^{\lfloor \log n \rfloor - 1}$. In the first method, each processor is assigned n/p records and 4 tapes to perform an external merge sort on this subset of n/p records. After p sorted runs have been produced by this parallel phase, a single processor merge-sorts them serially.

In the second method, the basic idea is that each processor performs a different phase of the serial-merge procedure. The i-th processor merges pairs of runs of size 2^{i-1} into runs of size 2^i for i=1, 2,...$\lfloor \log n \rfloor$. (Ideally, n is a power of 2 and log n processors are available.) A high degree of parallelism is achieved by using the output tapes of a processor as input tapes for the next processor so that as soon as a processor has completed 2 runs these runs can be read and merged by another processor.

3.2.2 Parallel Binary Merge Sort

In this section we describe a merge-sort algorithm which utilizes both parallelism during each phase and pipelining between the phases to enhance performance. This parallel algorithm is an adaptation of the Even's tape sorting algorithm. It assumes a multiprocessor system with a three-level memory hierarchy. Each processor is assumed to have enough local internal memory to hold three pages of the file being sorted. Two of the buffers are used to hold input pages and the remaining one is used to hold output page. Between the processors and the mass storage device holding the file to be sorted (i.e., a disk drive) is a shared memory which acts as a disk cache. Processors are assumed to have access to this shared memory through an interconnection device such as a banyan [Goke73] or a cross-point switch [Wulf72]. The algorithm does not

assume that the disk cache is large enough to hold the entire file to be sorted. Rather this disk cache is simply useful for improving performance of the algorithm. By using such an organization we show that the delay between processors can be shortened (no rewinding of the tapes is required) and that the 4p magnetic tapes requirement can be eliminated. In [Bora80a], a binary merge sort without pipelining of the phases was analyzed. The parallel binary sort algorithm presented below represents a significant improvement.

Execution of this algorithm is divided into three stages as shown in Figure 10. We assume that there are at least twice as many pages in the file being sorted as there are processors. The algorithm begins execution in a suboptimal stage in which sorting is done by successively merging pairs of longer and longer runs until the number of runs is equal to twice the number of processors. First, each of the p processors reads 2 pages and merges them into a sorted run of 2 pages. This step is repeated until all single pages have been read. If the number of runs of 2 pages is greater than 2*p, each of the p processors proceeds to the second phase of the suboptimal stage in which it repeatedly merges 2 runs of 2 pages into sorted runs of 4 pages until all runs of 2 pages have been processed. This process continues with longer and longer runs until the number of runs equals 2*p.

When the number of runs equals 2*p each processor will merge exactly two runs of length n/2p. This phase is called the optimal stage. At the beginning of the postoptimal stage the controller releases one processor and logically arranges the remainder as a binary tree (see Figure 10). During the postoptimal stage parallelism is employed in two ways. First, all processors at the same level of the tree execute concurrently. Second, pipelining is used between levels in a manner similar to the pipelined merge sort (described in the previous section) except that each processor outputs a single run rather than two. By pipelining data between levels of the tree, a parent is able to start its execution a single time unit after both its children (i.e., as soon as its children have produced one page). Therefore, the cost of the postoptimal stage will be a 2-page operation for each level of the tree plus the cost for the root processor to merge two runs of length n/2.

Analysis: If p=n/2, there is no suboptimal stage and the processor at the top of the binary tree waits log(n/2) units of time before it starts merging 2 runs of size n/2. During each unit of time, a processor will read two pages, merge them together and write out one page. Therefore, the algorithm terminates in $\log(n/2)C_p^3 + nC_p^1$ time.

If $p < n/2$, then during each of the $\log(n/2p)$ phases of the suboptimal stage each processor executes a total of n/p page operations (i.e., $n/2p\ C_p^1$ operations). In phase i the runs are one half the size of the runs of phase i+1, but each of the p processors performs twice as many merge operations in order to exhaust the runs. Thereafter, the top processor waits logp units of time before it starts merging two runs of length n/2. Therefore, the total execution time of the algorithm is

$$\underbrace{(n/p) * \log(n/2p) C_p^1}_{\text{suboptimal phase}} + \underbrace{\log p\, C_p^3}_{\text{waiting}} + \underbrace{n\, C_p^1}_{\text{merging}}$$

3.2.3 The Pipelined Selection Sort

Unlike the two previous algorithms, this algorithm does not produce longer runs during intermediate steps of its execution. Therefore, it does not require that the processors merge blocks longer than a single page. However, it is not as fast since its complexity is $O(n^2/p)$ compared to $O(n\log n/p)$. Nevertheless, we feel that it has other properties (in particular its simplicity) which make it worthwhile considering.

Basically, the algorithm is based on iterative selection. The maximum of n pages is determined, then the maximum of the remaining n-1 pages is determined and the operation is repeated until the last (i.e., the minimum) page is created. By "maximum" page we mean that page with the highest key values. To determine it, 2 pages are merged and the first page of the sorted run is kept. This page is then merged with a third page, and again the first page of the result run is kept. By repeating this process until all the source pages are exhausted, the maximum page is obtained.

Parallelism is introduced by having one processor assigned to each step of maximum selection. In other words, the first processor selects the maximal page among n pages, the second processor selects the maximal page among the remaining n-1 pages, etc. If enough processors are available, the algorithm performs optimally when p=n-1 processors are assigned to the sort operation. In this case, the processors are labeled P_1, P_2,..,P_{n-1} and logically organized as a pipeline. P_1 reads sequentially the source relation pages. During each C_p^1 time unit, it reads a new page, merges it with the page that was previously kept in its buffer and sends the lower page to P_2. After P_2 has received 2 pages, it starts processing in the same way and sends its lower page to P_3. As the pipeline is filled, the pages flow one at a time through the processors. When the

last page reaches P_{n-1}, P_i, for i=1, 2,..., n-2, contain the i-th page of the sorted relation. To complete the sort operation, P_{n-1} merges the 2 pages it has received and writes them out, in order, as the 2 last pages of the sorted relation. The time to sort in this case is

T = time for P_1 to read n pages, merge n-1 pairs of pages and write n-1 pages,

+ time for the last page to propagate through the pipeline.

= C_r (* for P_1 to read the first page *)
+ (n-1) C_p^1
+ (n-1) C_p^1
+ C_w (* for P_{n-1} to write the last page *)

$T = (2n-2)C_p^1 + C_r + C_w$

which is approximately equal to $(2n-1)C_p^1$.

Note that we have omitted the time for the first n-2 processors to write their result page. This is because P_i can write its result page while P_{i+1} reads the page P_i has previously written.

In the general case, when $p < n-1$, the algorithm requires multiple phases. Each phase repeats the basic linear pipeline algorithm, except that processor P_p must write out excess pages that no other processor can receive. During the first phase, this creates a bucket of n-p pages which is not sorted. On the other hand, the p pages residing in P_1, P_2,..., P_p constitute the first p pages of the sorted relation. For n=kp, the algorithm will require k phases with each phase producing p pages of the sorted relation. If n is not an exact multiple of p, then the only modification is for the last phase: if less than p pages are left then the last phase uses a shorter pipeline of length n mod p, and terminates in 2(n mod p)-2 C_p^1 page operations. The execution time of the algorithm is the sum of the execution times of the k phases, where the i-th phase takes:

$n - (i-1)p - 1 + (p)\ C_p^1$ time units

This sum is equal to

$p * k(k+1)/2 - k + k * (p)$

or in terms of n and p

$n^2/2p + 3n/2 - n/p$

For large n, and p less in order of magnitude than n this is of order $n^2/2pC_p^1$.

This algorithm has several advantages. In particular, it appears simple to implement and it seems that it would imply no storage overhead. Unlike the previous sorting algorithms, it does not require that the controller maintain

page tables for temporary relations or complex control tables for processor reassignment. Also, the algorithm can be implemented efficiently on a ring architecture, where the processors are connected by a bus that allows simultaneous transfers from a processor to its right neighbor.

3.3 Block Sorting Algorithms

For all the algorithms to be presented below, we will not assume the presence of a multiported disk cache as was done in some previous algorithms. The presence of such a disk cache makes the architecture not easily extensible. This is because the use of additional processors will require the disk cache to have additional ports. Hence, the disk cache is not assumed to be part of the architecture. Instead, all the sorting methods to be suggested will use p parallel processors, where each processor has attached to it enough secondary memory to accommodate M pages of the file to be sorted. The algorithms will also need additional workspace as will be discussed below. Altogether, Mp=n pages can be sorted. Since the architecture does not have a shared disk cache memory, processors will have to exchange records by sending messages via some interconnection network.

The basic idea behind all the algorithms to be presented in this section is as follows. Consider a sorting algorithm which uses only comparison-exchange steps. A comparison-exchange step requires the comparison of the sort values of two records and an exchange in the positions of these two records if they are found to be out of order. A local comparison-exchange step is a comparison-exchange step performed on two records in the primary memory of a single processor. A non-local comparison-exchange step is a comparison-exchange step performed on a record in a processor's primary memory and a record in another processor's primary memory. A block sorting algorithm is obtained by replacing every comparison-exchange step in a non-adaptive sorting algorithm consisting only of comparison-exchange steps with a merge-split step.

A merge-split step of order M merges two sorted runs M pages long and produces a sorted run 2M pages long which is then split into two parts. A local merge-split of order M is a merge-split of order M in which the two original and the two final sorted runs are both stored in the secondary memory associated with a single processor. A non-local merge-split of order M is a merge-split of order M in which the two original and the two final sorted runs are stored in the secondary memories associated with two different processors. A local merge-split of order 4 is shown in Figure 11a and a non-local merge-

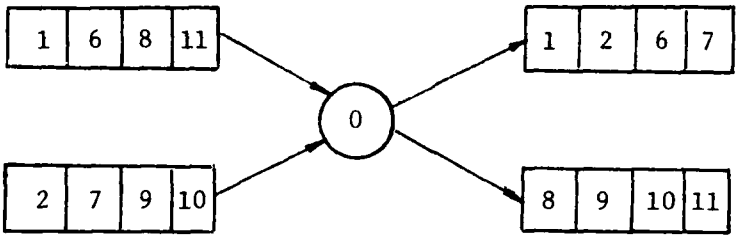


Figure 11a. A Local Merge-Split of Order 4

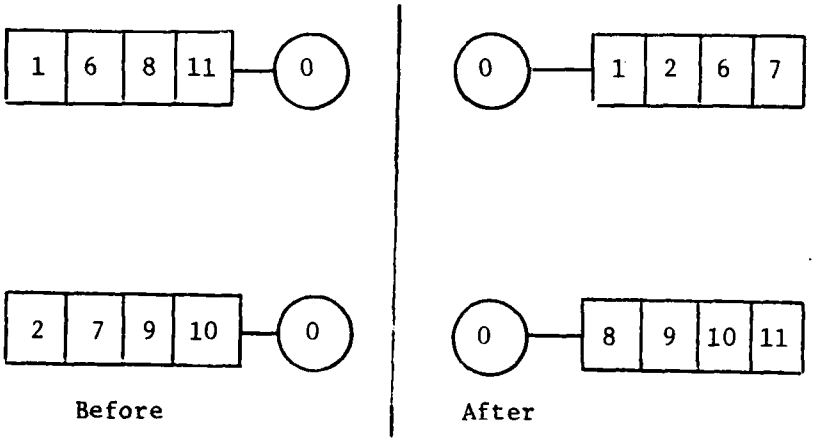


Figure 11b. A non-Local Merge-Split of Order 4

split of order 4 is shown in Figure 11b.

A block sorting algorithm is obtained by replacing every local (non-local) comparison-exchange step in a sorting algorithm consisting only of comparison-exchange steps with a local (non-local) merge-split step.

We will assume that a local merge-split step of order M is performed in the obvious way taking $2MC_p^1$ time units. However, there are two different techniques for performing a non-local merge-split and this leads to the two different classes of block sorting algorithms to be discussed below. If an algorithm consists only of local comparison-exchange steps there is only one corresponding block sort algorithm. An example of such an algorithm is the bitonic sorter of Batcher. However, if a sorting algorithm contains non-local comparison-exchange steps, there are two corresponding block sort algorithms because there are two different techniques for executing the non-local merge-split steps corresponding to the non-local comparison-exchange steps in the algorithm. An example of such an algorithm is the odd-even transposition sort.

In the first section below, we will present the block bitonic sort which consists only of local merge-split steps. Then, in the next sections, we will present block sorting algorithms which contain non-local merge-split steps. First, we will present a family of block sorting algorithms in which each non-local merge-split is performed by merge sorting [Baud78]. Finally, we will present a family of block sorting algorithms in whcih each non-local merge-split is performed as a series of local comparison-exchanges [Hsiao80].

3.3.1 Block Bitonic Sort

In Section 2.2, we presented Batcher's bitonic sorter which could sort 2p numbers with p processors in $\log^2 2p$ shuffle steps and 1/2(log2p+1)(log2p) local comparison-exchange steps. By replacing each local comparison-exchange step with a local merge-split step, we obtain the block bitonic sorter which can sort Mp pages of records with p processors in $\log^2$ 2p shuffle steps and 1/2(log2p+1)(log2p) local merge-split steps.

Because the block bitonic algorithm can process at most 2p blocks (runs) with p processors, a prepocessing stage is necessary when the number of pages to be sorted exceeds 2p. The function of this preprocessing stage is to produce 2p sorted blocks of size n/2p pages each. We have identified two ways of performing this preprocessing stage. The first is to use a parallel binary merge to create 2p sorted blocks (runs) of n/2p pages each. The second is to execute a bitonic sort in several phases with blocks of size 1, 2p, $(2p)^2$,...

until blocks of size n/2p pages are produced. We have analyzed both approaches and have discovered that the first approach is approximately twice as fast as the second for large n and relatively small p. Therefore, we present below only an analysis of the first.

The first part of the algorithm is identical to the suboptimal phase of the parallel binary merge and completes in $\frac{n}{p} \log (\frac{n}{2p})C_p^1$ time. The $\log^2 2p$ shuffle steps take $\log^2 2p(\frac{n}{p})C_s$ time, since each shuffle step requires each processor to transfer $(\frac{n}{p})$ pages of records to another processor. Each local merge-split is of order $\frac{n}{2p}$ and hence takes $\frac{n}{p} C_p^1$ time units. The total time of execution is therefore

$$(\frac{n}{p})\log(\frac{n}{2p})C_p^1 + (\frac{n}{p})\log^2 2pC_s + (\frac{n}{2p})(\log 2p)(\log 2p + 1)C_p^1$$

or

$$\frac{n}{p}(\log n + 1/2(\log^2 2p - \log 2p))C_p^1 + \frac{n}{p} \log^2 2pC_s$$

3.3.2 Merge-Sort Based Block Sorting

In these algorithms, a non-local merge-split of order M is performed as below. Consider the situation of Figure 12a, in which processor 0 has four sorted pages and processor 1 has four sorted pages. It will be seen that the non-local merge-split of order M will require 4M pages of secondary memory to be associated with each processor. In the example of Figure 12, M is four. Thus, each processor must have 16 pages of secondary memory associated with it. Only the contents of those pages of memory relevant to the discussion are shown in Figure 12 and the other pages are shown as being empty. The merge-split proceeds as follows. Processor 1 sends its four sorted pages to processor 0 which places these four pages in its memory. The situation of this point is shown in Figure 12b. Processor 0 will then merge sort these two sorted runs of four pages to produce a sorted run of eight pages as shown in Figure 12c. Finally, processor 0 will output the higher four pages of the sorted run back to processor 1 completing the non-local merge-split of order 4 as shown in Figure 12d. Clearly, the time for a non-local merge-split by this method is $2MC_s + 2MC_p^1$.

3.3.2.1 Merge-sort Based Block Odd-even Sort

This method is a generalization of the odd-even transposition sort [Knut73] which is illustrated in Figure 13. The p horizontal lines represent

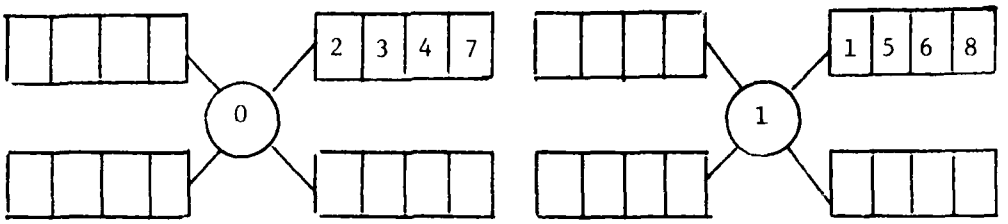


Figure 12a

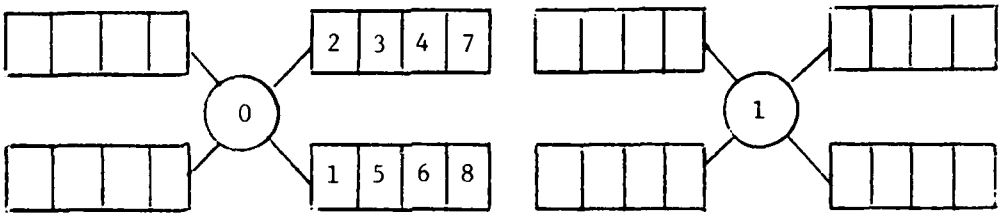


Figure 12b

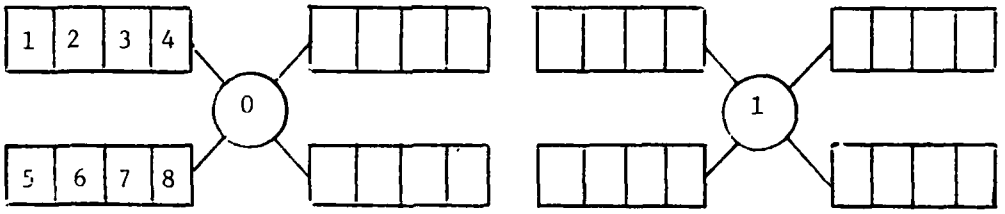


Figure 12c

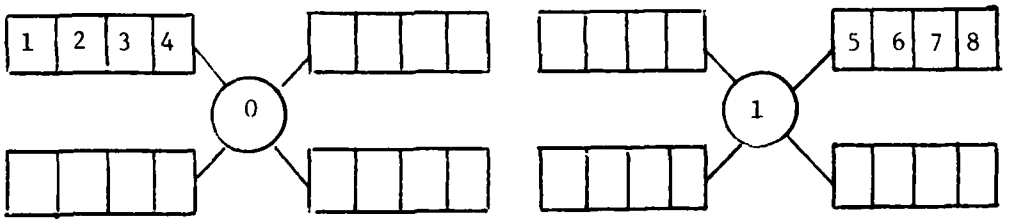


Figure 12d

Figure 12. Four Steps Involved in the Merge-Split of Order 4

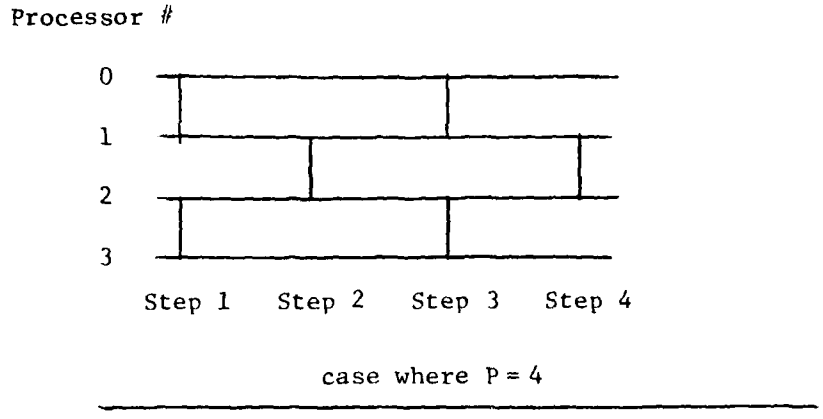


Figure 13. The ODD-EVEN Transposition Sort

the p processors, each of which contains one of the total of p elements to be sorted. Sorting proceeds in p steps. In the odd-numbered steps, parallel non-local comparison-exchanges occur between processors 0 and 1, 2 and 3,..., etc. In the even-numbered steps, parallel non-local comparison-exchanges occur between processors 1 and 2, 3 and 4, ..., etc. The generalization of this algorithm to a block odd-even sort would entail p parallel non-local merge-split steps. Furthermore, it requires a preprocessing stage to produce p sorted runs of size n/p each and this takes $(n/p)\log(n/p)C_p^1$ time units. Thus, the total time of execution is

$$(\frac{n}{p})\log(\frac{n}{p})C_p^1 + p(2MC_s + 2MC_p^1) = (\frac{n}{p}\log n - \frac{n}{p}\log p + 2n)C_p^1 + 2nC_s$$

3.3.2.2 Interconnection of Processors

It is easy to see that, in this algorithm, each Processor i, for $1 \leq i \leq p-2$, interacts directly only with Processors i+1 and i-1. Processor 0 only interacts with Processor 1, and Processor p-1 interacts only with Processor p-2. Therefore, for each processor, we need only connect directly to a maximum of two other processors, the one 'in front of' it, and the one 'behind' it. Figure 14 shows the nature of interconnections for various numbers of processors including an additional controller processor. We note, at this point, that there is no restriction on the number of processors, i.e., the value of p. P can be any positive number. Also, since each processor needs to be connected directly only to two others, we have a hardware structure which is easily extensible.

3.3.3 Comparison-Exchange Based Block Sorting

The main problem with merge-sort based block sorting is that it needs 4Mp pages of secondary memory in order to sort Mp pages of records using p processors. This large amount of secondary memory is necessitated by the method used for non-local merge-splitting. The authors [Hsia80] have discovered a new technique for merge-splitting which may be used to create block sorting algorithms which can sort Mp pages of records using p processors with only 2Mp pages of secondary memory. Furthermore, these block sorting algorithms have better time of execution than the block sorting algorithms described in the previous section.

In these algorithms, a non-local merge-split of order M is performed as follows. First, find the largest record (i.e., the record with the largest

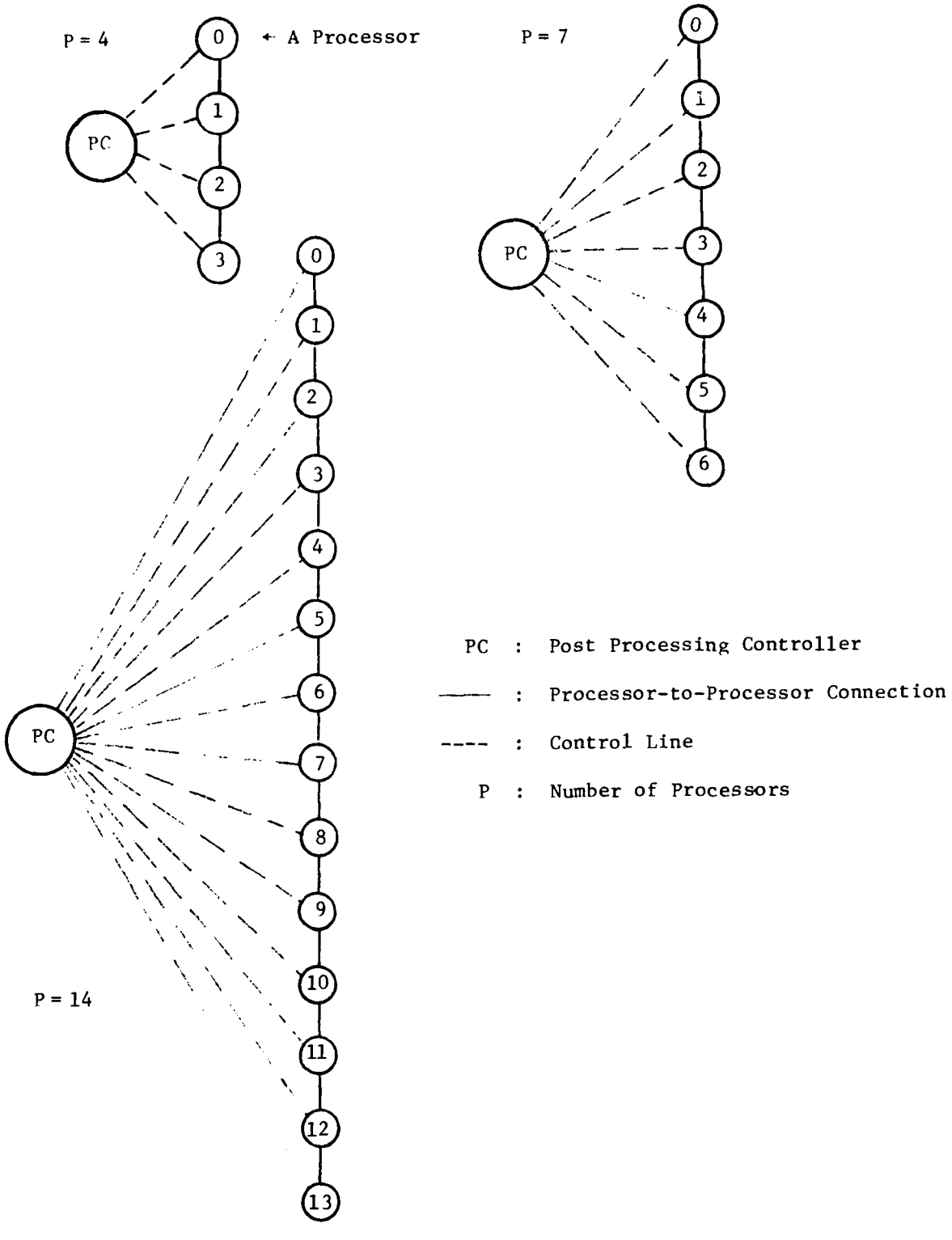


Figure 14. Interconnection of Processors for Odd-Even Sort

sort value for the sort attribute of those records in the same memory) in Processor 0's memory and also find the smallest record (i.e., the record with smallest sort value for the sort attribute) in Processor 1's memory. If the largest record in Processor 0's memory is less than or equal to the smallest record in Processor 1's memory, there is no need to exchange. Otherwise, we exchange the record in one memory with the record in the other memory. Next, find the second largest record in Processor 0's memory and the second smallest record in Processor 1's memory and compare the sort attributes of the two records. Once again, if the second largest record in Processor 0's memory is less than or equal to the second smallest record in Processor 1's memory, there is no need to exchange. Otherwise, the records are exchanged. The process continues until no more exchanging is needed. Thus, all M records in Processor 0's memory are smaller in sort values than the sort values of the M records in Processor 1's memory.

The entire process of exchanging records may be done as follows if we assume that the sorted records in processor 0's memory are in descending order of the sort values. At the same time, we assume that the sorted records in processor 1's memory are in ascending order of the sort values. Now, Processors 0 and 1 compare and exchange, if necessary, corresponding records, i.e., the first record of Processor 0 (the record with largest sort value in Processor 0's memory) with the first record of Processor 1 (the record with smallest sort value in Processor 1's memory), the second record of Processor 0 (the record with the second largest sort value in Processor 0's memory) with the second record of Processor 1 (the record with the second smallest sort value in Processor 1's memory), and so on. At the end of the exchange process, the sort values of all the M records in Processor 0's memory are smaller than any sort value of the M records in Processor 1's memory. Furthermore, the sort values of the M records in processor 0's memory and the sort values of the M records in processor 1's memory both form bitonic sequences. The aforementioned steps are illustrated in Figure 15. The idea is based on the discovery by Alekseyev [18] that in order to select the largest t elements out of 2t elements $< x_1, x_2, \ldots, x_{2t} >$, we may first sort $< x_1, x_2, \ldots, x_t >$ and then sort $< x_{t+1}, x_{t+2}, \ldots, x_{2t} >$ and then compare and interchange x_1 with x_{2t}, x_2 with $x_{2t-1}, \ldots, x_t$ with x_{t+1}. After records are exchanged between the memories in the manner described above, let each processor do a localized sort of records in its own memory on the basis of the sort values (see Figure 15 again).

This localized sorting may be done by merging the bitonic sequence start-

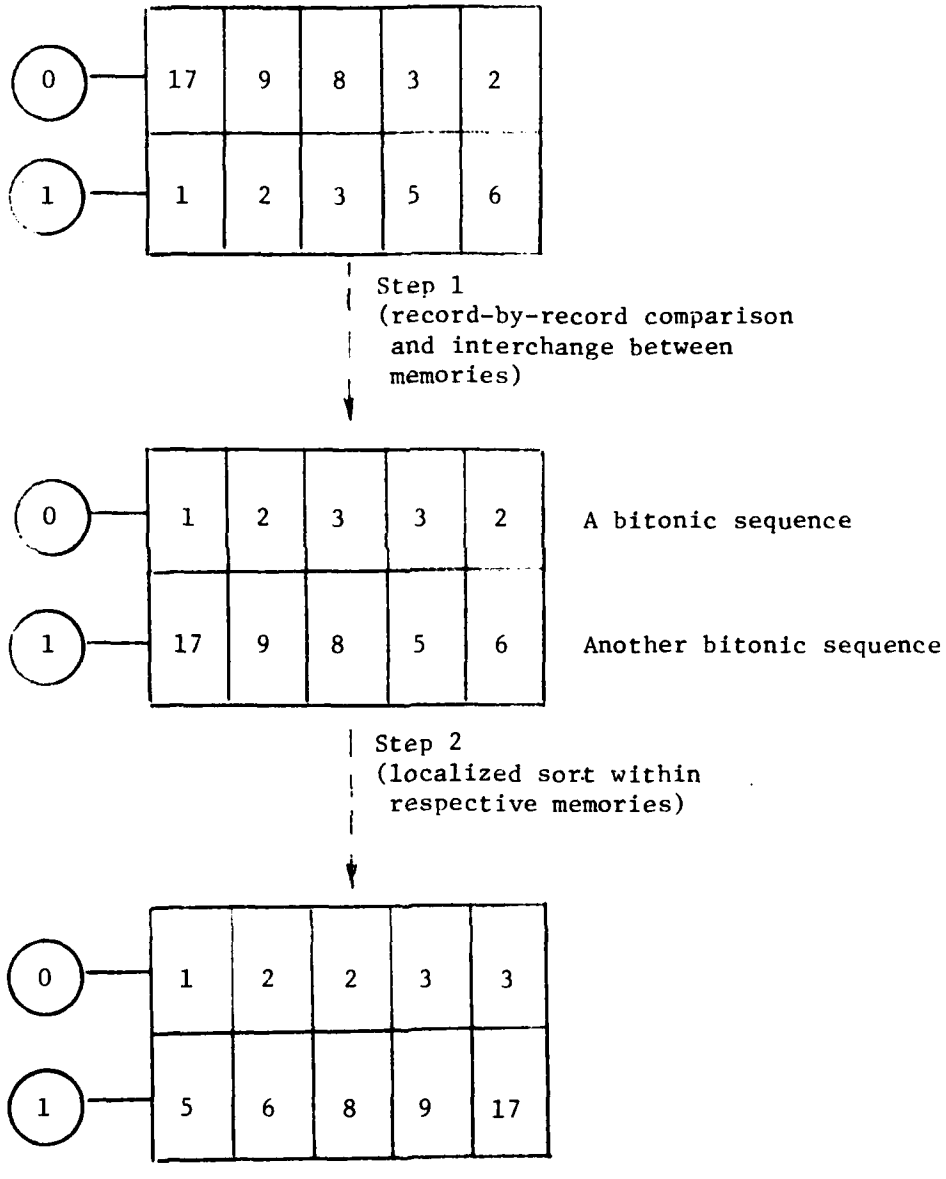


Figure 15. A Non-Local Merge-Split of Order 5 in Two Steps

ing from the two ends of the sequence taking MC_p^1 time. The compares and interchanges required for performing the merge-split take a worst case time of $2MCs + MC_p^2$ and an average case time of $MCs + (M/2)C_p^2$. Thus, the average case time for a merge-split of order M is

$$MCs + MC_p^1 + \frac{M}{2}C_p^2$$

which is better than the time for the merge-split of order M by the method of the previous section. Furthermore, this method for merge-splitting does not require 4M memory to be associated with each processor. In fact, only one additional page is needed per processor for the merge-splitting making a total of (M+1) pages of memory per processor. However, the preprocessing stage of the block sorting algorithms to be described requires 2M pages of memory to be associated with each processor. Thus, the block sorting algorithms to be described require a total of 2Mp pages of secondary memory as opposed to the 4Mp pages required by the algorithms of the previous section.

In the next section, we shall analyze two algorithms [Hsia80] for sorting which use the above techniques for merge-splitting. The first is a generalization of the odd-even transposition sort. The second is a generalization of a modification of the bitonic sort.

3.3.3.1 Comparison-Exchange Based Block Odd-even Sort

As discussed in Section 3.3.2.1, the method will require a preprocessing stage which takes

$$(n/p)\log(n/p)C_p^1 \text{ time units}$$

Furthermore, it requires p parallel non-local merge-splits which takes

$$p(MCs + MC_p^1 + \frac{M}{2}C_p^2) \text{ time units}$$

Therefore, the total time of execution is

$$(n/p \log n - n/p \log p + n)C_p^1 + nC_s + \frac{n}{2}C_p^2$$

In Figure 16, we present an example in which p parallel processors sort Mp records for p=4 and M=5.

3.3.3.2 Comparison-Exchange Based Block Modified Bitonic Sort

The bitonic sorter, as discussed in Section 2.1.2, may be used to sort n keys with n/2 processors, where each of the n/2 processors is connected to four other processors. A modification of this sorter is one which may be

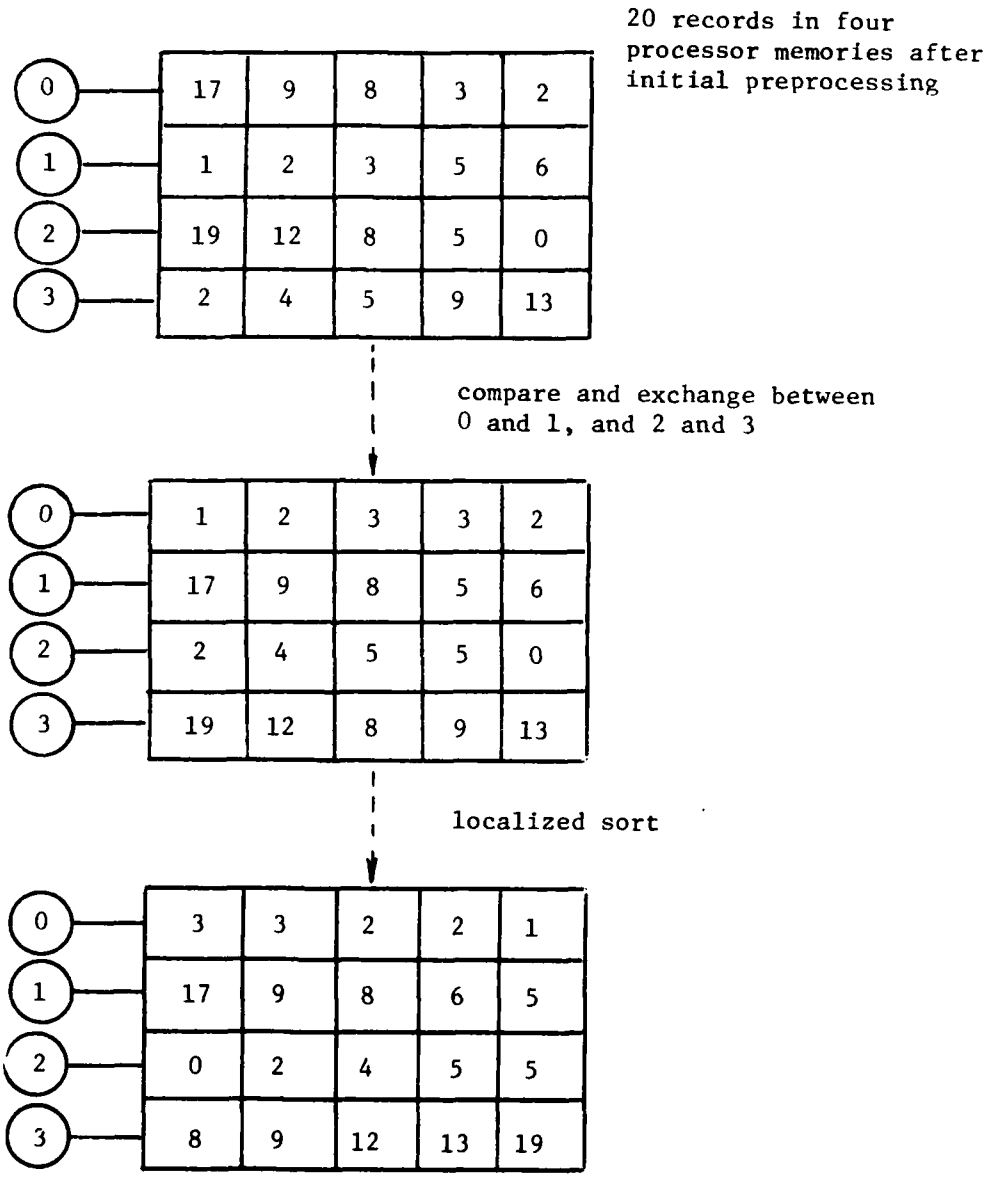


Figure 16. Sorting 20 Records Using the Block Odd-Even Sort

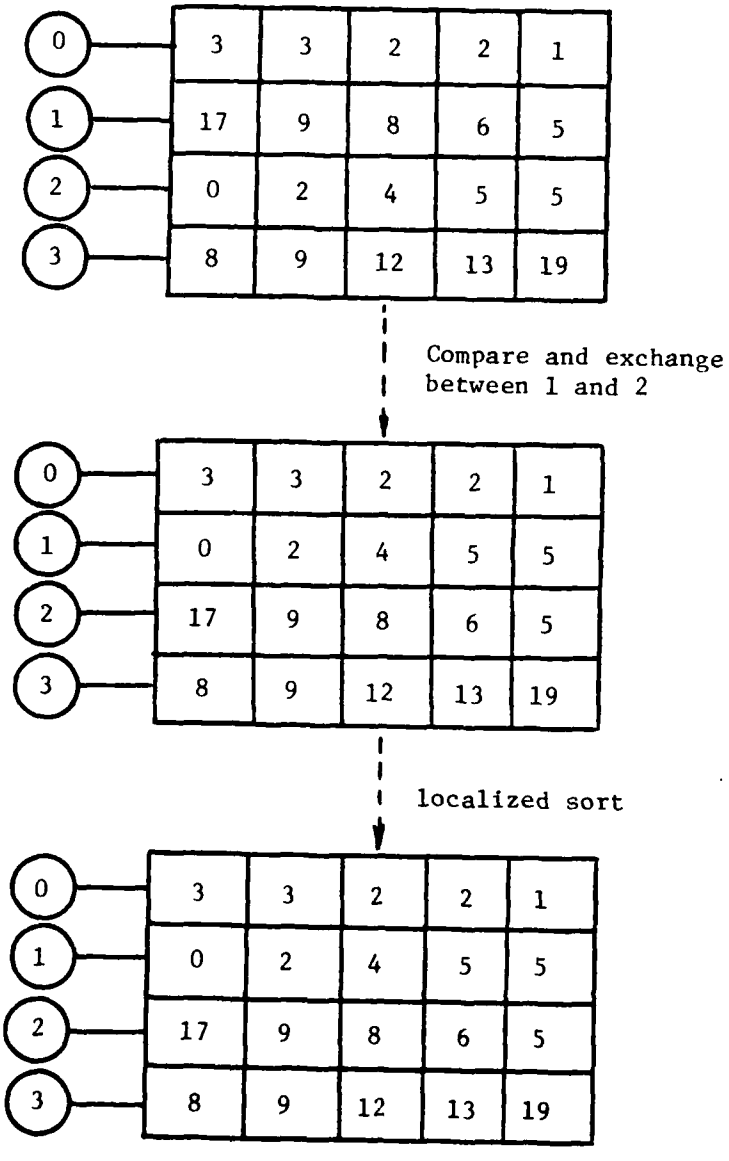


Figure 16. Continued

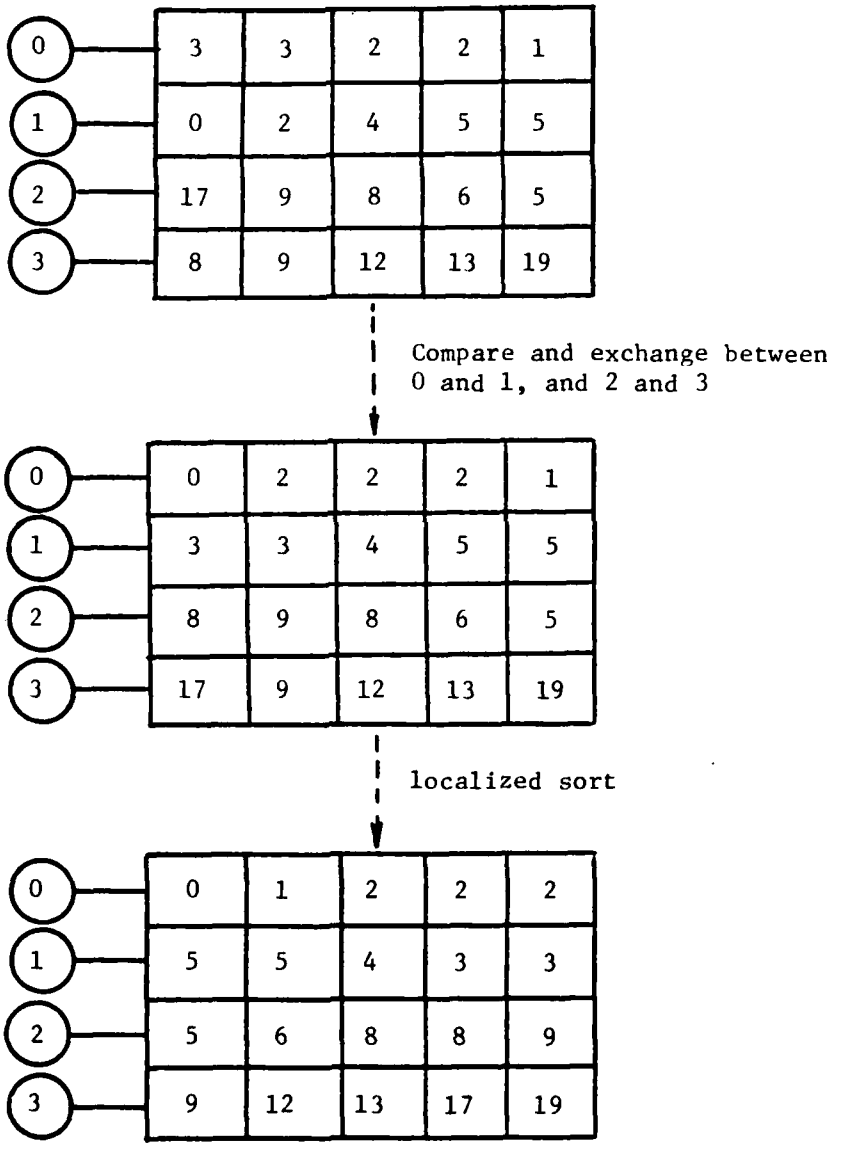


Figure 16. Continued

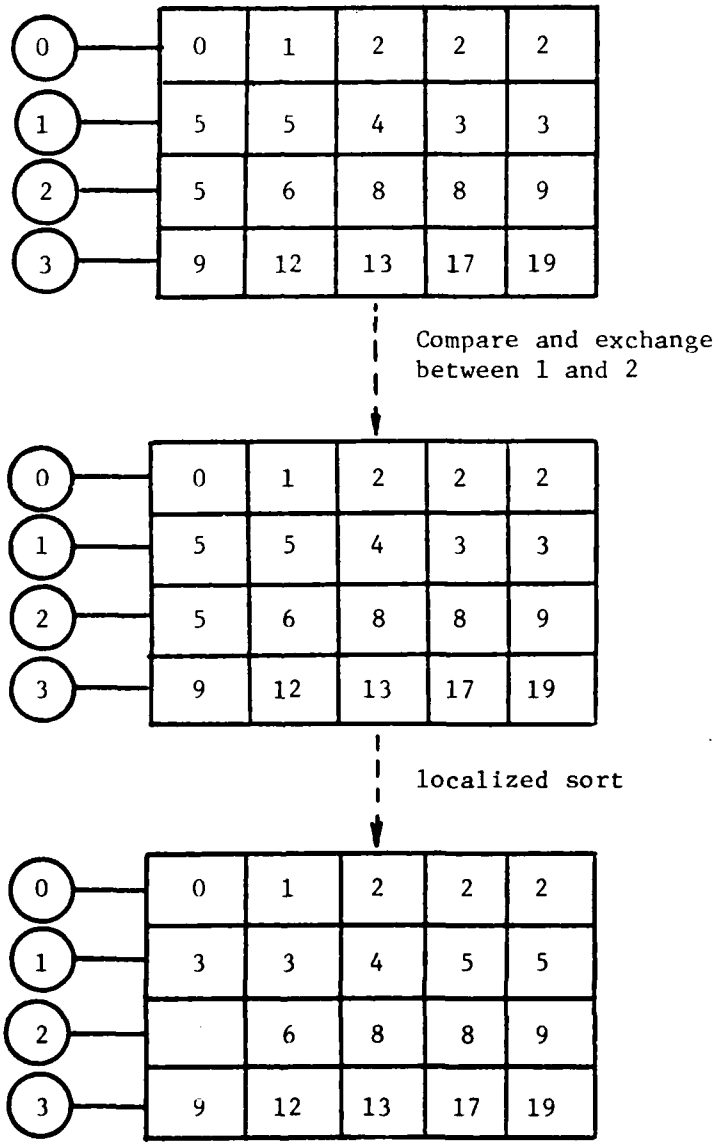


Figure 16. Continued

used to sort n keys with n processors, where each of the n processors is connected to only two other processors [Hsia80]. A description of how sorting proceeds in the modified bitonic sorter for four keys is graphically shown in Figure 17. As can be seen from there, there are four ($\log^2 4$) stages in the algorithm. Each stage consists of a shuffle followed by zero or more parallel non-local comparison-exchanges. The first stage consists of a shuffle followed by no non-local comparison-exchanges. The second stage consists of a shuffle followed by parallel non-local comparison-exchanges between processors 0 and 1 and between processors 2 and 3. The shuffles are indicated by the non-vertical arrows. The vertical arrows indicate the non-local comparison-exchanges. Thus, the vertical arrow in stage 2 from processor 3 to processor 2 indicates a non-local comparison-exchange between processors 2 and 3. The direction of the arrow indicates that, after the non-local comparison-exchange, processor 3 will receive the smaller-valued record and processor 2 will receive the larger-valued record. The actions to be performed in the remaining two stages are similarly indicated in the figure.

We see that whereas the bitonic sorter had only local comparison-exchanges (see Figure 6), the modified bitonic sorter has only non-local comparison-exchanges. The advantage of the modified bitonic sorter is that it requires only two interconnections per processor as opposed to four needed in the bitonic sorter. The advantage will exist even when we generalize the modified bitonic sorter in an external sorting algorithm. The disadvantage of the modified bitonic sorter is that it may store only one key per processor as opposed to two keys per processor for the bitonic sorter. This disadvantage is overcome when we generalize the modified bitonic sorter to an external sorting routine where each processor has enough memory to store several records. This explains the motivation for modifying the bitonic sorter before generalizing it.

As before, the generalization consists of replacing all non-local comparison-exchanges with non-local merge-splits. Therefore, the generalized algorithm consists of $\log^2 p$ shuffles and $1/2(\log p)(\log p+1)$ non-local merge-splits. Furthermore, it requires a preprocessing step requiring

$$(n/p)\log(n/p)C_p^1 \text{ time units}$$

Thus, the total execution time is

$$\frac{n}{p}\log(\frac{n}{p})C_p^1 + (\log^2 p)\frac{n}{p}C_s + (\frac{1}{2})(\log p)(\log p + 1)(\frac{n}{p}C_s + \frac{n}{p}C_p^1 + \frac{n}{2p}C_p^2)$$

or $(\frac{n}{p}\log n + \frac{n}{2p}\log^2 p - \frac{n}{2n}\log p)C_p^1 + ((\frac{3n}{2p})\log^2 p + \frac{n}{2p}\log p)C_s + ((\frac{n}{4p})\log p + (\frac{n}{4p})\log^2 p)C_p^2$

time units.

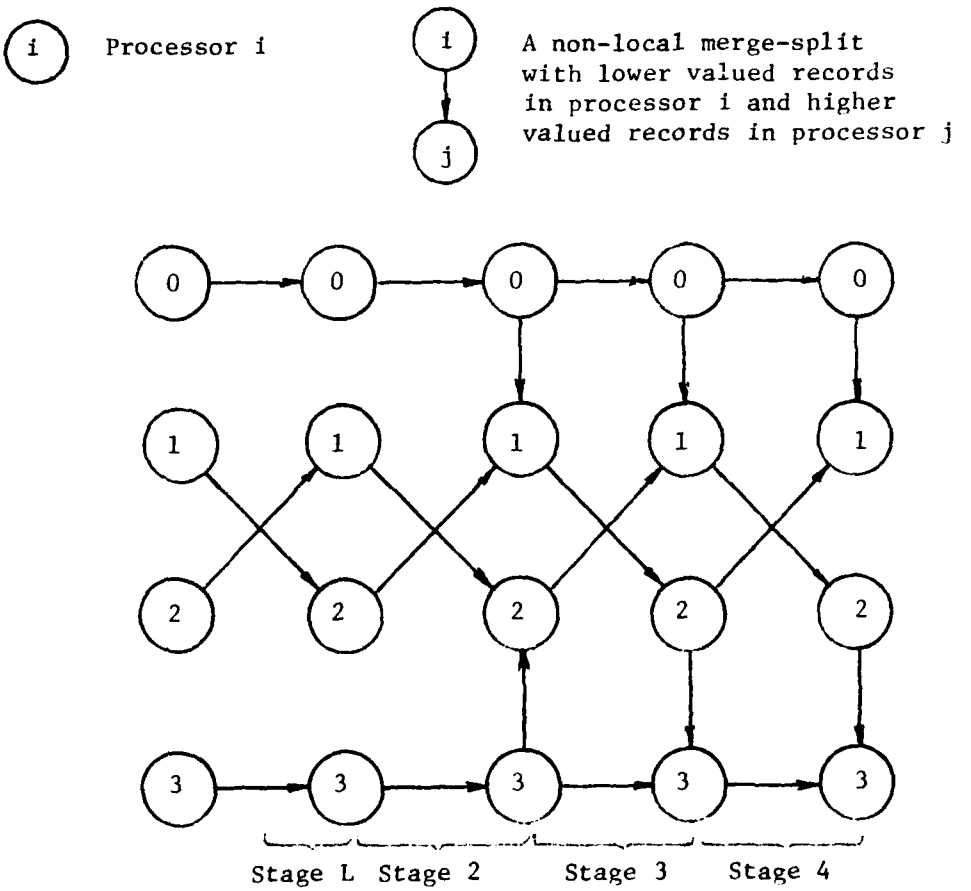


Figure 17. Four Stage Modified Bitonic Sort with Four Processors

3.3.3.3 Interconnection of Processors

From the algorithm, it is easy to see what kind of interconnections are needed between processors. First, to be able to provide the perfect shuffle, each processor must be connected directly to its shuffle processor which is calculated as follows.

Let ibin = binary notation of decimal i,
ibin' = ibin after left circular shift by one bit, and
j = decimal equivalent of ibin'. Then
connect Processor i to Processor j.

For P = 8, we have the following interconnections.

Processor 0 is connected to no other processor
Processor 1 is connected to Processor 2
Processor 2 is connected to Processor 4
Processor 3 is connected to Processor 6
Processor 4 is connected to Processor 1
Processor 5 is connected to Processor 3
Processor 6 is connected to Processor 5
Processor 7 is connected to no other processor.

These connections need only be one-way connections. That is, for example, Processor 6 needs to be able to send records to Processor 5, but Processor 5 need not be able to send records to Processor 6.

Also, to provide for the merge-split operations, we need that Processor i, $0 \leq i \leq (P-1)$, be connected to Processor $(i+1)$ if i is even, or to Processor $(i-1)$ if i is odd. These connections, unlike the previous ones, are two-way connections. The algorithm needed to decide on all interconnections (so as to be able to provide for both Shuffles and Merge-Splits) is shown below.

```
i = 0
while i < P
do ibin = i in binary
   ibin' = ibin after left circularly shifting by one bit
   j = decimal equivalent of ibin'
   connect Processor i to Processor j
      (if j is different from i)
       by using a one-way connection

   if i is even
      then connect Processor i to Processor (i + 1)
         using a two-way connection
      else connect Processor i to Processor (i - 1)
         using a two-way connection
      i = i + 1
end
```

The layout of processors and their interconnections for various values of p are shown in Figure 18.

4. SUMMARY AND FUTURE RESEARCH DIRECTIONS

In this paper we have presented both a description of the operation of a variety of internal and external parallel sorting algorithms and an analysis of the performance of each algorithm. While the internal sorting algorithms have only limited application to the problems of sorting in a database environment this line of research has resulted in the development of a number of important theoretical results especially Preparata's shared-memory multiprocessor enumeration sort which achieves the optimal level of performance: log n for n processors.

An important issue which needs further investigation is whether the performance criteria by which parallel sorting algorithms have been previously evaluated are general enough. Clearly, assuming that the number of processors is as large as the number of elements to be sorted and counting the number of parallel comparisons as the main cost factor is not satisfactory. Communication costs and, in the case of external sorting, I/O costs must be incorporated in a comprehensive analytical model which is general enough to accommodate a wide range of multiprocessor architectures.

One important aspect of a parallel algorithm is the amount of control it requires to be executed correctly and efficiently. In particular, an adequate granularity of synchronization must be determined for an algorithm to perform its task correctly but without making processors wait for significant amounts of time. On an SIMD machine, algorithms that are very synchronous by nature can be implemented in a straightforward manner. (An example of such an algorithm is the odd-even transposition sort described in Section 2.3). Implementing the same algorithms on an MIMD machine requires significantly more overhead for controlling the processors. However, synchronous algorithms can be generalized so that the period of time required for synchronous tasks becomes longer. For example, replacing a comparison-exchange of two elements by a merge-split of two blocks of data reduces significantly the level of synchronization required by a sorting algorithm.

Unfortunately, the MIMD approach implies more overhead for control of the algorithms. One or several processors must coordinate the actions of the other processors which are executing a task. To accomplish this task, complex tables must be stored and maintained in the controller's memory, and control

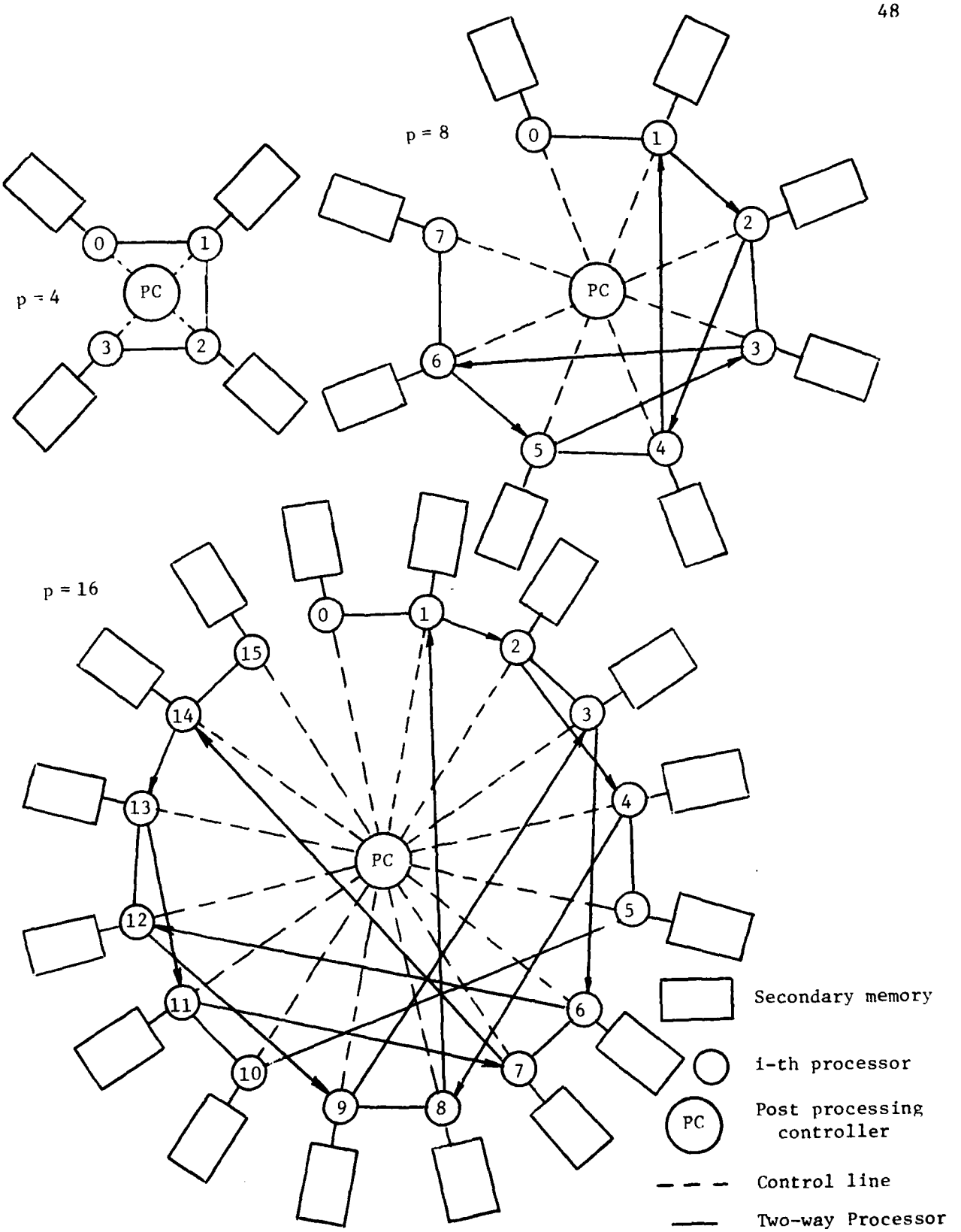


Figure 18. Interconnection of Processors for Sorting Using Modified Stone Sort

messages must be exchanged between the controller and the processors. We have identified four components of control cost that should be added to parallel algorithm cost evaluation:

(1) The overhead for reassignment of processors between phases of the algorithm execution.

(2) The synchronization of processors at initiation and termination of the algorithm, of a phase (e.g., the optimal phase in the parallel binary merge), and of a step within a phase.

(3) The allocation of pages to processors.

(4) The storage and maintenance of temporary files created during the algorithm execution (for example, a merge sort will require creating a temporary relation to store the output runs of each phase).

Each of these components require communication overhead (for the exchange of messages between the controller and the processors) and processing overhead (for the controller to look-up and update its tables). Techniques for evaluating both types of overhead need to be developed.

5. REFERENCES

[Batc68] Batcher, K.E., "Sorting Networks and Their Applications," 1968 Spring Joint Computer Conference, AFIPS Proceedings, Vol. 32, 1968.

[Baud78] Baudet G. and Stevenson, D., "Optimal Sorting Algorithms for Parallel Computers," IEEE-TC, Vol. c-27, No. 1, January 1978.

[Bora80a] Boral, H. and DeWitt, D.J., "Processor Allocation Stratgies for Multiprocessor Database Machines," ACM TODS, Vol. 6, No. 2, June 1981, page 227-254; Also Computer Sciences Technical Report #368, University of Wisconsin, October 1979.

[Bora80b] Boral, H., DeWitt, D.J., Friedland, D. and Wilkinson, W.K., "Parallel Algorithms for the Execution of Relational Database Operations," Computer Sciences Technical Report #402, University of Wisconsin, October 1980. Also submitted to ACM TODS.

[Chan76] Chandra, A.K., "Maximal Parallelism in Matrix Multiplication," IBM Report RC. 6193, Watson Research Center, Yorktown Heights, N.Y., October 1976.

[Csan76] Csanky, L., "Fast Parallel Matrix Inversion Algorithm," SIAM Journal of Computing, Vol. 5, No. 4, December 1976.

[Desp78] Despain, A.M. and Patterson, D.A., "X-TREE: A Tree Structured Multi-processor Computer Architecture," Proceedings of the Fifth Annual Symposium on Computer Architecture, 1978.

[DeWi79a] DeWitt, D.J., "Query Execution in DIRECT," Proceedings of the ACM SIGMOD 1979 International Conference on Management of Data, May 1979.

[DeWi79b] DeWitt, D.J., "DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems," IEEE-TC, Vol. c-28, No. 6, June 1979.

[Even74] Even, S., "Parallelism in Tape Sorting," CACM, Vol. 17, No. 4 April 1974.

[Flan77] Flanders, P.M., Hunt, D.J., Reddaway, S.F. and Parkinson, D., "Efficient High Speed Computing with the Distributed Array Processor," Symposium on High Speed Computer and Algorithm Organization, 1977, University of Illinois.

[Fran80] Franklin, M.A., "VLSI Performance Comparison of Banyan and Crossbar Communications Networks," Proceedings of the Workshop on Interconnection Networks for Parallel and Distributed Processing, April 1980.

[Gavr75] Gavril F., "Merging with Parallel Processors," CACM, Vol. 18, No. 10, October 1975.

[Goke73] Goke, G.R. and Lipovski, G.J., "Banyan Networks for Partitioning Multiprocessor Systems," Conference Proceedings of the First Symposium on Computer Architecture, Dec. 1973.

[Good80] Goodman, J.R. and Despain, A.M., "A Study of the Interconnection of Multiple Processors in a Database Environment," Proceedings of the 1980 International Conference on Parallel Processing, August 1980.

[Hawt80] Hawthorn, P. and DeWitt, D.J., "Performance Evaluation of Database Machines," Submitted to IEEE-TC.

[Hirs78] Hirschberg, D.S., "Fast Parallel Sorting Algorithms," CACM, Vol. 21, No. 8, August 1978.

[Hsia80] Hsiao, D.K. and Menon, M.J., "Parallel Record-Sorting Methods for Hardware Realization," Technical Report OSU-CISRC-TR-80-7, The Ohio State University, Columbus, Ohio, July 1980.

[Knut73] Knuth, D.E., The Art of Computer Programming, Vol. 3, Sorting and Searching, Addison-Wesley, 1973.

[Lin76] Lin, C.S., Smith, D.C.P. and Smith, J.M., "The Design of a Rotating Associative Memory for Relational Database Applications," ACM TODS, Vol. 1, No. 1, March 1976.

[Mull75] Muller, D.E. and Preparata, F.P., "Bounds for Complexity of Networks for Sorting and for Switching," JACM, April 1975.

[Nass79] Nassimi, David and Sahni, S., "Bitonic Sort on a Mesh Connected Parallel Computer," IEEE-TC, Vol. c-27, No. 1, January 1979.

[Ozka75] Ozkarahan, E.A., Schuster, S.A. and Smith, K.C., "RAP - An Associative Processor for Data Base Management," Proceedings 1975 NCC, Vol. 45, AFIPS Press, Montvale, N.J.

[Peas68] Pease, M.C., "An Adaptation of the Fast-Fourier Transform for Parallel Processing," JACM, Vol. 15, April 1968.

[Prep78a] Preparata, F.P., "New Parallel Sorting Schemes," IEEE-TC, Vol. c-27, No. 7, July 1978.

[Prep78b] Preparata, F.P. and Sarwate, D.V., "An Improved Parallel Processor Bound in Fast Matrix Inversion," IPL, Vol. 7, No. 3, April 1978.

[Rose69] Rosenfeld, J.L., "A Case Study in Programming for Parallel Processors," CACM, Vol. 12, No. 12, December 1969.

[Russ78] Russel, R., "The Cray-1 Computer System," CACM, Vol. 41, No. 2, 1978.

[Sieg77] Siegel, H.J. "The Universality of Various Types of SIMD Machine Interconnection Networks," Proceedings of the Fourth Annual Symposium on Computer Architecture, March 1977.

[Slot70] Slotnick, D.L., "Logic Per Track Device," Advances in Computers, Vol. 10, J. Tou, ed., Academic Press, N.Y., 1970.

[Ston71] Stone, H.S., "Parallel Processing with the Perfect Shuffle," IEEE-TC, Vol. c-20, No. 2, February 1971.

[Su75] Su, S.Y.W. and Lipovski, G.J., "CASSM: A Cellular System for Very Large Data Bases," Proceedings International Conference on Very Large Data Bases, September 1975.

[Thom77] Thompson, C.D. and Kung, H.T., "Sorting on a Mesh Connected Parallel Computer," CACM, Vol. 20, No. 4, April 1977.

[Vali75] Valiant, L.G., "Parallelism in Comparison Problems," SIAM Journal of Computing, Vol. 3, No. 4, September 1975.

[Wulf72] Wulf, W.A. and Bell, C.G., "C.mmp - A Multi-mini-processor," Proceedings of Fall Computer Conference, pp. 765-777, 1972.